



POLITECNICO DI TORINO

Master degree course in Ingegneria Informatica

Master Degree Thesis

Quantum Safe TPM

Supervisor

Prof. Antonio Lioy
Ing. Grazia D'Onghia

Candidate

Davide PALATRONI

April 2025

Questa tesi è dedicata a tutte le persone che mi hanno accompagnato e sostenuto lungo il mio percorso fino a oggi. Alla mia famiglia, in particolare a mia madre Raffaella, a mio padre Ettore, e alle mie sorelle Aurora e Gioia: grazie per avermi supportato e sopportato con amore, pazienza e fiducia, anche nei momenti più difficili. A Chiara, la mia ragazza, che anche da lontano è riuscita a trasmettermi serenità e a farmi sentire sempre il suo sostegno. A chi è presente oggi, in questo giorno speciale, e a chi non può esserci più ma continua a vivere nei miei ricordi. Ai miei compagni di viaggio durante l'università, in particolare Riccardo e Andrea: la vostra presenza e calma mi ha aiutato ad affrontare ogni esame con più leggerezza. A chi mi conosce

*da sempre e oggi è qui a
festeggiare con me. Al mio
amico Davide, con cui,
nonostante tutto, ogni sabato,
da ormai 8 anni, ci
ritroviamo come sempre. Ai
miei zii e ai miei nonni, che
non mi hanno mai fatto
mancare nulla, regalandomi
affetto e valori. Grazie a tutti
voi e grazie alle esperienze
vissute insieme: se oggi sono
la persona che sono, lo devo
anche a voi.*

Contents

1	Introduction	10
1.1	Context	10
1.2	Use case	11
1.3	Solution	12
1.4	Thesis Structure	12
2	Post-Quantum Cryptography	14
2.1	Introduction	14
2.2	Evolution of Quantum Computing	15
2.3	Threat to Modern Cryptography	16
2.3.1	Symmetric Key Cryptography	16
2.3.2	Cryptographic Hash Functions	16
2.3.3	Public Key Cryptography	17
2.4	Challenges in Advancing Quantum Computing	17
2.5	Mitigate Quantum Threat	18
2.6	Post-Quantum Algorithms	18
2.6.1	Lattice-based Cryptography	19
2.6.2	Multivariate-based Cryptography	19
2.6.3	Hashed-based Signatures	20
2.6.4	Code-based Cryptography	20
2.6.5	Super Singular Elliptic Curve Isogeny Cryptography	20
2.7	Choice of Algorithm: SPHINCS-SHAKE-256f-simple	21
2.7.1	Overview of SPHINCS	21
2.7.2	Design Considerations	21
2.7.3	Variants and Selection Criteria	21
2.7.4	Strengths of SPHINCS-SHAKE-256f-simple	21
2.7.5	Proposed Solution: Integration into an fTPM	22

3	Trusted Computing	23
3.1	Introduction	23
3.1.1	Motivation and Background	23
3.1.2	Conceptual Framework	23
3.1.3	Trusted vs Trustworthy	23
3.1.4	Relevance in Cybersecurity	24
3.2	Trusted Computing Group	24
3.3	Foundation of Trust: The Root of Trust	24
3.3.1	Definition and Purpose	24
3.3.2	Static vs Dynamic Trust	25
3.3.3	Types of Root of Trust	25
3.4	Trusted Platform Module	25
3.4.1	Role in Trusted Computing	25
3.4.2	Key Security Features	26
3.4.3	Limitations of TPM	27
3.4.4	TPM Versions	27
3.4.5	TPM Implementations	28
3.5	Trusted Execution Environment (TEE)	29
3.5.1	Introduction	29
3.5.2	TEE Security Guarantees	30
3.5.3	Remote Attestation and Secure Storage	30
3.5.4	Lifecycle Management and Hybrid Trust Model	31
3.5.5	TEE Architecture	31
3.5.6	Applications	33
3.6	ARM TrustZone	33
3.6.1	Introduction	33
3.6.2	Processor Mode Separation	33
3.6.3	Exception Levels and Execution Contexts	34
3.6.4	World Identification and the NS Bit	34
3.6.5	Memory and Peripheral Isolation	35
3.6.6	Virtual Memory	35
3.6.7	Secure Boot and Chain of Trust	35
3.6.8	Secure Storage and Replay Protection	36
3.6.9	World Switching and Secure Monitor	36
3.6.10	Interrupts in ARM TrustZone	37
3.6.11	Key Management in ARM TrustZone	38
3.6.12	Shortcomings of ARM TrustZone	38
3.6.13	Comparison between technologies	39
3.7	Remote Attestation	41
3.7.1	Introduction	41
3.7.2	Attestation Workflow	41
3.7.3	PCR Usage in PC Clients	42

4	Technologies Used	43
4.1	OP-TEE	43
4.1.1	OP-TEE Architecture	43
4.1.2	Communication Between REE and TEE in OP-TEE	43
4.1.3	Security and Protection of Trusted Applications	46
4.1.4	OP-TEE APIs and Advanced Features	46
4.2	Tpm2-tss	47
4.2.1	Tpm2-tss Architecture	47
4.2.2	Role of Tpm2-tss	48
4.2.3	Benefits of Tpm2-tss	48
4.3	Tpm2-tools	48
4.3.1	Features of Tpm2-tools	48
4.4	Liboqs	49
4.4.1	Supported Algorithms	50
5	Design and Implementation	52
5.1	Design Choices	52
5.1.1	General Architecture	53
5.2	Technological Choices	54
5.3	Project Structure	54
5.4	Ms-tpm-20-ref	55
5.4.1	Project Architecture and Structure	55
5.4.2	Execution in OP-TEE	55
5.4.3	Cryptographic Function Extension	55
5.4.4	TPM Hierarchies	55
5.5	Optee_fTPM	56
5.6	Tpm2-tss and Tpm2-tools	56
5.6.1	Reasons for modifications	56
5.6.2	Technical Details	57
5.7	Considerations	58
6	Tests and Results	59
6.1	Introduction	59
6.2	Configuration of Test Environment	59
6.3	Test Methodology	60
6.3.1	Functional Tests	61
6.3.2	Performance Tests	61
6.4	Results	61
6.4.1	Key and Signature Generation Performance Analysis	62
6.4.2	OP-TEE Overhead	63
6.5	Comparison and Final Observations	64

7	Conclusions	66
A	User Manual	68
A.1	Requirements	68
A.2	Build Steps	68
A.2.1	Install Dependencies	68
A.2.2	Build Project	68
A.3	ftPM Use	69
A.3.1	Example of commands	69
B	Developer Manual	71
B.1	Requirements	71
B.2	Installation	71
B.2.1	Install Dependencies	71
B.2.2	Cloning of repositories	71
B.3	Project Configuration	72
B.3.1	Build	72
B.3.2	Liboqs	72
B.3.3	Optee_os	74
B.3.4	Ms-tpm-20-ref	74
B.3.5	Optee_ftpm	86
B.3.6	Tpm2-tss and Tpm2-tools	87
	Bibliography	92

Chapter 1

Introduction

This chapter provides an initial definition of the problem, explaining why and how quantum computers threaten communications and data integrity, identifying the main targets of quantum attacks, and outlining the solution proposed to mitigate these risks.

1.1 Context

The evolution of quantum technology presents one of the most significant challenges for modern cybersecurity. Once quantum computers reach sufficient maturity, they can break many of the cryptographic algorithms currently in use, compromising the security of protected communications and data. To address this potential threat, the scientific community has initiated the development of post-quantum cryptographic algorithms designed to withstand quantum attacks.

As quantum computing capabilities advance, most cryptographic algorithms currently in use face to risk of becoming vulnerable. Algorithms such as Rivest-Shamir-Adleman (RSA) and Elliptic Curve Cryptography (ECC) could be easily broken by a sufficiently powerful quantum computer, primarily due to the efficiency of quantum algorithms like Shor's algorithm [1]. Shor's algorithm enables the factoring of large numbers in polynomial time, rendering asymmetric key algorithms insecure. This vulnerability necessitates the transition to new cryptographic schemes that are resistant to quantum attacks.

A fundamental concept in securing modern computing systems is the Root of Trust (RoT). The RoT is a foundational security mechanism that ensures the integrity and authenticity of a device's operations by establishing a chain of trust. It consists of hardware, firmware, or software components that provide a secure foundation for cryptographic operations. Secure boot, cryptographic key management, and device identity verification are among the key features enabled by the RoT, ensuring that a system remains protected. Hardware-based RoT implementations, such as those integrated into TPMs, play a critical role in anchoring trust within a device. These mechanisms ensure that only authenticated and unaltered software is executed, mitigating risks posed by both classical and quantum threats.

All computational devices relying on asymmetric encryption, particularly embedded devices, are at risk of quantum attacks. In this context, the Trusted Platform Module (TPM) plays a crucial role in device security by providing secure cryptographic functions and ensuring data integrity and confidentiality [2]. Specifically, TPMs provide several critical security functionalities. They generate and store cryptographic keys in a secure, tamper-resistant manner, preventing unauthorized access and ensuring the confidentiality of sensitive data. Through encryption and decryption mechanisms, TPMs protect information from being intercepted or altered by unauthorized entities. Digital signature verification ensures that authentication processes remain intact and data integrity is maintained. Additionally, TPMs enable remote attestation, allowing a device to prove its trustworthiness by generating cryptographic proofs of its hardware and software state. They

also support measured boot, which verifies the integrity of system components at startup, ensuring that only trusted software is executed. These features collectively provide a robust security foundation, especially in embedded environments where resources are constrained and security is paramount.

TPMs make extensive use of asymmetric key algorithms for secure authentication, encryption, and digital signatures [3]. However, with the advent of quantum computers. These traditional cryptographic methods will no longer be secure. Consequently, TPMs must adopt quantum-resistant (QR) algorithms to continue protecting sensitive data within embedded environments [4]. This transition is particularly challenging due to the computational and memory constraints typical of embedded systems, necessitating efficient implementations of post-quantum cryptographic primitives.

The adoption of these new algorithms is not straightforward for several reasons. QR primitives require a paradigm shift and must be translated into a new set of performance and memory constraints due to the limited resources available in embedded devices. Additionally, it is not possible yet to obtain a physical TPM, but software-based TPM is a viable solution [3]. By emulating a TPM in software, it is possible to integrate QR primitives into existing TPMs securely.

Recognizing the urgency of this transition, the National Institute of Standards and Technology (NIST) has taken a proactive approach by evaluating and standardizing post-quantum cryptographic algorithms. After extensive analysis and multiple rounds of assessment, NIST has selected four candidates for standardization [5]. These selected algorithms are designed to address different security requirements while balancing computational efficiency and resource constraints, making them viable for a variety of applications, including embedded systems and TPMs. Each has its advantages and limitations. For instance, SPHINCS+ provides a highly secure signature but can take up to two minutes to generate a single signature, making it unsuitable for real-time applications [2]. Therefore, careful selection of the appropriate algorithm is necessary based on specific requirements.

This thesis explores key challenges in integrating post-quantum algorithms within TPMs, analyzing security implications, hardware and software constraints, and strategies for a smooth transition to quantum-resistant encryption.

1.2 Use case

A primary concern for embedded devices is how to implement security mechanisms, as these devices form the foundation of security in more complex systems. This leads to the concept of Trusted Computing, which describes technologies that establish trust in local and remote computing systems by leveraging trustworthy components, known as trust anchors, to ensure system integrity [6]. The Trusted Computing Group (TCG) has made significant efforts toward standardizing trusted computing practices, defining a set of requirements compiled in the GlobalPlatform Specifications [7].

A key component of trusted computing is the Trusted Execution Environment (TEE), a separate and tamper-resistant processing environment that operates independently from the Rich Execution Environment (REE) [8]. The REE represents the standard operating environment where general-purpose applications run, often exposed to potential security threats due to its openness and accessibility. Unlike the REE, the TEE is designed to provide a higher level of security by ensuring that sensitive operations are executed in an isolated and protected space. The TEE is used for sensitive operations that require higher security levels, while the REE handles non-sensitive operations. TEE ensures critical security features such as remote attestation and measured boot, both of which are central to this study.

One approach to implementing a TEE in an embedded device is through a secure and isolated

processing mode, as seen in ARM TrustZone technology. TrustZone is widely used in modern mobile devices and embedded systems. Typically, embedded devices have small processors, limited computational resources, minimal I/O capabilities, small storage, and stringent power constraints [9]. In contrast, many post-quantum cryptographic algorithms are computationally intensive and require larger sizes, making them less suitable for embedded environments.

The hardware platform chosen for this study is the Xilinx ZynQ UltraScale+, a System-on-a-Chip (SoC) that integrates a traditional processor with an FPGA. Specifically, the processor under consideration is the ARM Cortex-A53, which utilizes ARM TrustZone to implement a TEE within embedded systems.

1.3 Solution

There are multiple approaches to implementing a TPM. For example, discrete hardware TPMs and integrated TPMs provide strong protection against physical attacks but are difficult to modify and update [3]. Consequently, software-based TPM emulation is a more practical option for development and testing, as it enables rapid experimentation without reducing the lifespan of non-volatile (NV) memory [3].

The technology selected for TEE implementation is ARM TrustZone, developed by ARM in 2002 [10]. TrustZone enables a system-wide security approach across various devices. It partitions system resources into two distinct environments: the Secure World (SW), where sensitive operations are executed and correspond to the TEE, and the Normal World (NW), where general applications run and correspond to the REE [11]. This separation guarantees security while executing sensitive applications. Additionally, the transition between these two worlds required a dedicated system call, further enhancing security.

Applications are also categorized accordingly: Trusted Applications (TA) execute within the Secure World, while Client Applications (CA) run in the Normal World and can invoke sensitive data only through TA calls [10].

A practical first step toward quantum-resistant TPMs is to integrate PQ algorithms into a firmware TPM (fTPM), which operates as a Trusted Application within TrustZone. This approach leverages TrustZone’s security features to protect critical operations and sensitive data.

Using an fTPM offers several advantages over a physical TPM. Firstly, an fTPM is more flexible and can be easily updated. Unlike physical TPMs, which require hardware modifications to be upgraded, fTPMs can integrate new cryptographic algorithms dynamically through firmware updates. Furthermore, an fTPM can be deployed across a wide range of devices without requiring hardware changes, making it a scalable and cost-effective solution. Additionally, it facilitates backward compatibility, easing the transition to next-generation TPMs.

By integrating PQ algorithms into an OP-TEE-based fTPM, this study aims to provide an advanced security solution that anticipates the future challenges posed by quantum computing. This approach ensures the protection of critical data and operations in an increasingly digital and quantum-aware world.

1.4 Thesis Structure

This work focuses on the integration of post-quantum (PQ) algorithms within TPMs and their application in embedded environments. The second chapter will discuss the threats posed by quantum computers, their impact on cybersecurity, and the solution proposed by NIST. The third chapter will introduce the concept of trusted computing, the role of the Trusted Computing Group, and the implementation of the Trusted Execution Environment (TEE). The fourth chapter will describe the technologies used to counteract the quantum threat, followed by an explanation of

the design choices made in this project. The last two chapters will focus on testing and conclusions. Additionally, a manual is included at the end, serving as a reference both for developers and for individuals who wish to utilize the proposed solution.

Chapter 2

Post-Quantum Cryptography

This chapter focuses on post-quantum cryptography, providing an overview of its context and importance. It begins by delving into the history and development of quantum computing, offering a foundational understanding of the technology. The chapter then explains the quantum threat and the potential risks posed by quantum computers to current cryptographic systems and explores how these advancements challenge traditional cryptography. Finally, it introduces the countermeasures being developed to address this threat, including the emerging field of quantum-resistant algorithms and their role in securing communication in a post-quantum era.

2.1 Introduction

In today's digital world, technological advancements in electronic communications have made security and privacy essential concerns. The need to protect data integrity, confidentiality, authentication, and non-repudiation has placed cryptography at the heart of modern information security.

Cryptography, whose name originates from the Greek words for "hidden" and "writing", is the practice of securing information during transmission and storage [12]. Its primary purpose is to prevent unauthorized access and tampering, ensuring that data remains private and intact. This discipline supports fundamental security principles such as the CIA triad, which stands for Confidentiality, Integrity, and Availability [13]. At its core, cryptography relies on encryption, a process that converts readable data, known as plaintext, into an unreadable format called ciphertext. This transformation is achieved using a secret key, which varies depending on the encryption scheme. In symmetric encryption, the same key is shared between communicating parties and must be kept confidential. In asymmetric encryption, a keypair enhances security by allowing encryption and decryption to be handled separately.

The rapid evolution of quantum computing presents a major challenge to traditional cryptographic systems. Originally theorized by Richard Feynman in 1982 [14], quantum computing has since grown into a field that could potentially disrupt modern security protocols. Quantum algorithms such as Shor's and Grover's have demonstrated the ability to solve complex mathematical problems far more efficiently than classical computers [15]. Specifically, Shor's algorithm can factor in large numbers and break widely used encryption methods, while Grover's algorithm can speed up search processes significantly. These advancements raise serious concerns about the security of current cryptographic standards, particularly those that rely on mathematical problems once considered computationally infeasible.

Although large-scale quantum computers have not yet been fully realized, ongoing research suggests that their development is only a matter of time. Some experts believe that recent breakthroughs indicate they could become viable shortly [16]. If this happens, many existing security systems, including those used in Internet of Things (IoT) infrastructures, could be at risk. The

ability of quantum computers to break encryption could compromise data confidentiality and integrity on a massive scale, creating vulnerabilities in critical communication systems.

To address this challenge, researchers are actively developing new cryptographic techniques resistant to quantum attacks. This emerging field, known as post-quantum cryptography, focuses on designing algorithms that can withstand the power of quantum computing. This chapter explores the implications of quantum threats, their impact on current security protocols, and the innovative solutions being developed to safeguard communications in a post-quantum era.

2.2 Evolution of Quantum Computing

Quantum computing was first conceptualized by Richard Feynman [14], who envisioned a machine capable of simulating natural physical phenomena. While initially theoretical, this field gained momentum towards the end of the 20th century when researchers like Peter Shor [1] and Lov Grover [17] demonstrated its potential in cryptography and search algorithms.

Unlike classical computing, which processes data using bits that exist as either 0 or 1, quantum computing operates with Qubits or quantum bits. These Qubits differ significantly from classical bits because they can exist in multiple states at once through a phenomenon called *superposition* [18, 19]. This property allows quantum computers to process vast amounts of information simultaneously, vastly increasing computational efficiency.

Another fundamental principle of quantum computing is entanglement. When two Qubits become entangled, their states remain intrinsically linked regardless of the physical distance between them [20]. A change in one Qubit instantaneously affects its entangled partner [12]. This unique characteristic enhances the computational power of quantum systems, enabling them to perform highly complex calculations with unprecedented speed.

Quantum computers leverage these properties to solve certain problems exponentially faster than classical computers. A theoretical system with n Qubits can execute 2^n operations in parallel [12], making it particularly effective in areas where classical algorithms struggle. Table?? summarizes the differences between Qubits and classical bits that permit quantum computers to improve their performances.

Feature	Classical Bits	Qubits
State	Can only be 0 or 1	Can be 0, 1 or a superposition of both
Physical Unit	Typically transistors or electrical circuits	Quantum particles
Underlying Principle	Based on binary logic	Based on quantum mechanic
Errors and decoherence	Rarely affected by external interference	Sensitive to decoherence and quantum errors
Measurable Information	Always deterministic (0 or 1)	The result is probabilistic until measured
Scalability	Easier to scale	Challenging to scale due to quantum phenomena
Computational Speed	Limited by classical principles	Can offer exponential advantages for certain problems

Table 2.1: Comparison between classical bits and Qubits

2.3 Threat to Modern Cryptography

The capabilities of quantum computing pose a significant risk to cryptographic systems currently in use. Many encryption protocols depend on mathematical problems that are infeasible for classical computers to solve but could be efficiently tackled by quantum algorithms.

Shor's algorithm [1], for instance, enables the rapid factorization of large numbers, threatening the security of public key encryption methods like RSA. Similarly, Grover's [17] algorithm accelerates search operations, reducing the security of symmetric encryption by cutting the time required to break an encrypted key in half. These breakthroughs demonstrate how quantum computing could undermine encryption methods that form the backbone of modern cybersecurity.

If quantum computers reach a sufficient level of stability and scalability, they could break widely used cryptographic protocols, endangering secure communications, online banking, and digital authentication. This growing threat has led researchers to explore alternatives that can resist quantum attacks.

2.3.1 Symmetric Key Cryptography

Symmetric-key cryptography (SKC) relies on a single shared key for both encryption and decryption. While this approach is computationally efficient, it presents challenges in securely distributing the key between parties, a problem that led to the development of asymmetric cryptography.

SKC algorithms fall into two main categories: stream ciphers and block ciphers. Stream ciphers encrypt messages one bit at a time, processing data as a continuous stream, whereas block ciphers encrypt fixed-size blocks of data. Some of the most widely used symmetric key algorithms include AES, IDEA, Blowfish, and RC4.

Despite its efficiency, SKC is vulnerable to certain attacks, such as the known-plaintext attack (KPA), where an attacker obtains pairs of plaintext and ciphertext to reconstruct the encryption key [13]. The emergence of quantum computing further exacerbates this issue. In 1996, Lov Grover introduced a quantum algorithm that can search an unsorted database in approximately the square root of the required operations. While Grover's algorithm does not completely break symmetric encryption, it effectively reduces the brute-force complexity of the key search, making encryption methods with smaller key sizes more vulnerable [17].

However, symmetric cryptography can remain secure against quantum attacks if key sizes are sufficiently large. According to NIST, the Advanced Encryption Standard (AES) is considered quantum-resistant when implemented with key sizes of 192 or 256 bits. AES-256, in particular, offers strong protection against quantum threats.

2.3.2 Cryptographic Hash Functions

Cryptographic hash functions are mathematical algorithms that convert input data or messages of arbitrary size into fixed-length output, called a hash or digest. For a hash function to be effective, it must meet several key requirements. First, it should be computationally efficient, producing the hash value quickly. Second, it must be deterministic, meaning the same input always produces the same output. Lastly, it must be resistant to collisions, where two different outputs produce the same hash value, which should be computationally infeasible to achieve [13].

Quantum computing presents a serious challenge to hash functions, much like it does to symmetric cryptography. Grover's algorithm can significantly speed up the process of finding collisions in a hash function, reducing the time required to find two different inputs that produce the same hash. Combined with the principles of the birthday paradox, this greatly increases the risk of hash function vulnerabilities [21].

To maintain security against quantum threats, hash functions must have a sufficiently large output size. A function designed to provide b -bit security against classical attacks would need an output length of at least $3b$ -bits to remain secure against quantum algorithms [12]. Increasing the digest length is one of the most effective countermeasures. As a result, modern hash functions such as SHA-2 and SHA-3, when configured with longer outputs, are considered resilient to quantum attacks[12].

2.3.3 Public Key Cryptography

Public key cryptography (PKC) relies on key pairs, consisting of a public key that can be openly shared and a private key that must be kept secure [13]. PKC is widely used for both encryption and digital signatures.

For encryption, the public key is used to encode messages so that only the corresponding private key can decrypt them, ensuring confidentiality. Digital signatures, on the other hand, use the private key to sign a message digest, allowing recipients to verify the authenticity and integrity of a document using the public key.

The security of PKC is based on hard mathematical problems, primarily the Factorization Problem and the Discrete Logarithm Problem (DLP). The Factorization Problem relies on the difficulty of breaking down large numbers into their prime components, while the Discrete Logarithm Problem involves solving modular exponentiation equations. Modern encryption schemes, such as RSA and Diffie-Hellman (DH), depend on these problems to ensure security.

However, quantum computing threatens the foundations of PKC. Shor's algorithm [1], designed for quantum computers, can efficiently solve both the Factorization Problem and the Discrete Logarithm Problem [22]. If large-scale quantum computers become practical, they could break RSA encryption and other public-key systems by solving these problems exponentially faster than classical computers.

Although quantum computers capable of breaking PKC do not yet exist, research into quantum algorithms has demonstrated their potential. In 2001, IBM successfully implemented Shor's algorithm on a small 7-qubit quantum computer, factoring in the number 15 [23]. While this was a limited demonstration, it proved the feasibility of quantum attacks on encryption.

The growing threat of quantum computing has led the cryptographic community to seek new encryption methods that can resist quantum attacks. As quantum technology advances, the need to transition toward quantum-resistant encryption is becoming increasingly urgent.

2.4 Challenges in Advancing Quantum Computing

Despite its potential, quantum computing still faces several technical challenges that must be overcome before it can pose a widespread threat to encryption. Stability and scalability are among the most pressing issues. QuBits are highly sensitive to external disturbances, such as temperature fluctuations and electromagnetic interference, which can disrupt their quantum state. To maintain coherence, quantum computers must operate at extremely low temperatures, close to absolute zero [24].

Another significant obstacle is the high error rate associated with quantum computations. Unlike classical bits, which maintain stable states, QuBits are prone to errors due to decoherence, requiring sophisticated error correction mechanisms to ensure reliable results. Additionally, the development of large-scale quantum computers with enough QuBits to break modern encryption remains an ongoing challenge, though experts estimate that within a decade there is a one-in-six chance of breaking RSA-2048, increasing to a fifty percent likelihood within fifteen years [25].

Given these uncertainties, researchers emphasize the importance of proactively developing quantum-resistant encryption before quantum computers reach their full potential. The urgency of this transition cannot be overstated, as waiting until quantum computers become powerful enough to break existing security protocols could leave critical systems vulnerable.

2.5 Mitigate Quantum Threat

In response to the growing risks posed by quantum computing, governments, research institutions, and cybersecurity organizations have intensified their efforts to develop post-quantum cryptography (PQC). These efforts focus on designing cryptographic algorithms that remain secure even against large-scale quantum attacks.

One of the most significant initiatives in this area is the National Institute of Standards and Technology (NIST) Post-Quantum Cryptography Standardization Project, launched in 2016. The goal of this project is to evaluate and standardize new cryptographic algorithms that can withstand quantum attacks. After multiple rounds of analysis, NIST has selected several promising quantum-resistant encryption schemes based on mathematical problems that are believed to be difficult even for quantum computers.

In 2022, NIST concluded the standardization process by selecting four algorithms deemed suitable for resisting quantum attacks:

- *CRYSTALS-Kyber*: Designed for general encryption purposes
- *CRYSTALS-Dilithium*: Designed for digital signatures
- *SPHINCS+*: Designed also for digital signatures, but using a different mechanism from CRYSTALS-Dilithium
- *FALCON*: Another digital signature algorithm, although no draft standard was provided for it.

In addition to developing new cryptographic standards, some governments and organizations are adopting hybrid encryption models, which combine classical and quantum-resistant encryption methods to ensure long-term security while maintaining compatibility with existing systems.

2.6 Post-Quantum Algorithms

Post-quantum cryptography aims to create cryptographic systems that remain secure against both quantum and classical computers while seamlessly integrating with existing communication protocols and networks [26]. Post-quantum cryptography relies solely on classical cryptographic principles, avoiding the use of quantum effects [27]. To address the potential threats posed by quantum computers, researchers are designing new cryptographic algorithms resistant to quantum attacks, known as post-quantum algorithms. These algorithms are grounded in mathematical problems believed to be difficult to solve, even for quantum computers. Notably, they are immune to solutions based on the Hidden Subgroup Problem (HSP) [12], which is the key concept of modern public key cryptographic schemes.

The primary challenge for post-quantum algorithms is to ensure long-term security while maintaining resilience against advancements in future computing technologies. Currently, five main categories of post-quantum implementations have been developed:

- *Lattice-based cryptography*
- *Multivariate-based cryptography*

- *Hashed-based signatures*
- *Code-based cryptography*
- *Super Singular Elliptic Curve Isogeny Cryptography*

2.6.1 Lattice-based Cryptography

Lattice-based cryptography refers to a broad category of cryptographic constructions that incorporate lattices into their design or security proofs. Lattice-based cryptography relies on challenging geometric problems associated with the structure of lattices in multidimensional space [28]. These problems, such as finding the shortest vector in a lattice or reducing short vectors, are considered extremely difficult to solve, even for quantum computers. The security of lattice-based cryptosystems primarily hinges on the complexity of two core problems: the *Short Integer Solution* (SIS) and the *Learning With Errors* (LWE) problems [20].

Several notable algorithms in this category include Kyber, NewHope, FrodoKEM, and FALCON. These algorithms offer significant security benefits and provide a balanced trade-off between performance, key size, and security guarantees [2].

- *Kyber* is based on LWE problem and is renowned for its efficiency and ease of implementation on standard hardware
- *NewHope* also relies on LWE and related lattice problems, demonstrating strong resistance to attacks and effectively securing communications
- *FrodoKEM* is another LWE-based algorithm, valued for its simplicity and robust security. However, it requires more computational resources compared to other lattice-based algorithms.
- *CRYSTALS-Dilithium* is a digital signature algorithm whose security is based on the computational complexity of solving lattice problems within modular lattices.[29].
- *FALCON*, short for "*Fast-Fourier Lattice-based Compact Signatures over NTRU*", leverages the lattice structure of NTRU combined with the Fast Fourier Transform (FFT) to enhance performance. Its primary benefit lies in the compact size of the signatures it produces, making it particularly well-suited for scenarios involving high-volume data transmission [30].

These characteristics make lattice-based cryptographic algorithms some of the most promising options for post-quantum security.

2.6.2 Multivariate-based Cryptography

Multivariate cryptographic schemes leverage the inherent difficulty of solving systems of multivariate polynomial equations over a finite field, a problem that has been proved to be NP-hard [20]. By capitalizing on the complexity of algebraic problems, these algorithms achieve secure encryption. Notable examples of multivariate cryptography include the Hidden Field Equations (HFE) and Rainbow systems. While these schemes are often faster in encryption and decryption compared to traditional methods, they tend to require larger keys.

- *Hidden Field Equations* (HFE) relies on polynomial equations to construct a system that resists attacks. Though it is designed to be robust, its computational complexity poses challenges
- *Rainbow* is widely recognized for generating secure digital signatures. It is more efficient than many other multivariate approaches, but ongoing research aims to further enhance its security and implementation

2.6.3 Hashed-based Signatures

Hash-based signature schemes rely on cryptographic hash functions, which are considered secure against quantum attacks, and require fewer security assumptions compared to number-theoretic signature schemes. This minimizes the security requirements needed to develop robust signature schemes [20]. Algorithms such as SPHINCS+ utilize combinations of hash functions to produce attack-resistant digital signatures. These schemes provide strong security while avoiding reliance on complex mathematical problems, making them simpler to analyze and implement.

- *SPHINCS+* combines hash trees with advanced hash techniques to construct a secure digital signature system. It is noted for its efficiency and quantum-resistance. Although it has slower signing operations and larger signature sizes, it compensates with a very small public key size and fast verification times [2].
- *Extended Merkle Signature Scheme* (XMSS) uses Merkle trees to generate digital signatures. Designed for simplicity and safety, XMSS ensures strong resistance to potential future attacks.

2.6.4 Code-based Cryptography

Code-based encryption is founded on the principles of error-correcting codes [12]. Classic cryptosystems in this domain include the McEliece (1978) and Niederreiter (1986) cryptosystems. In these systems, encryption involves the use of linear code, represented by generating or check matrix, which is masked through a series of transformations [27]. The strength of these cryptosystems lies in the computational difficulty of decoding a general linear code, a problem known to be NP-hard.

Code-based algorithms, such as McEliece and NTS-KEM, are highly resistant to quantum attacks. While they offer exceptional security, their key sizes are significantly larger than those used in traditional cryptographic systems.

- *McEliece* utilizes Goppa codes to build a highly secure encryption system. Although it requires large keys, it is widely regarded as one of the most robust algorithms against quantum threats.
- *NTS-KEM* is another code-based algorithm that balances security with computational efficiency. It is designed to be more practical for implementation compared to other code-based systems.

These features make code-based encryption a reliable option for post-quantum cryptography, especially for applications where robust security is a priority. *NTS-KEM* is another code-based algorithm, which offers a balance between security and computational efficiency. It is designed to be more practical in terms of implementation than other code-based systems.

2.6.5 Super Singular Elliptic Curve Isogeny Cryptography

This approach is relatively new, with development beginning in the early 21st century. It leverages the properties of isogenies between elliptic curves to construct cryptographic schemes. The foundation of these systems lies in the computational complexity of determining the isogeny between two elliptic curves over a finite field [27].

A major advantage of isogeny-based cryptosystems is their small key sizes compared to other post-quantum cryptographic schemes. However, their low operating speed currently limits their practicability [27]. Efforts are underway to improve their efficiency and make it more viable.

- *Supersingular Isogeny Key Encapsulation* (SIKE) relies on the difficulty of finding isogeny between supersingular elliptic curves. However, recently it was cracked by experts outside NIST with a classic computer [31].

- *Commutative Supersingular Isogeny Diffie-Hellman* (CSIDH) is another isogeny-based scheme designed for secure cryptographic key exchange. While it is simple and secure in concept, further research is needed to enhance its efficiency and implementation

2.7 Choice of Algorithm: SPHINCS-SHAKE-256f-simple

2.7.1 Overview of SPHINCS

For the post-quantum integration within the firmware TPM, the algorithm chosen is SPHINCS+, specifically the SHAKE-256f-simple variant. SPHINCS+ is one of the four digital signature schemes officially standardized by NIST as part of its Post-Quantum Cryptography project. What sets SPHINCS+ apart from many other post-quantum algorithms is its foundation on hash-based constructions, rather than more complex mathematical problems such as lattices or error-correcting codes. This choice of primitive reflects a conservative and well-understood security approach, which is particularly attractive when long-term trust and reliability are essential.

2.7.2 Design Considerations

A key characteristic that makes SPHINCS+ well-suited for this application is its stateless design. In contrast to stateful hash-based signature schemes, SPHINCS+ does not require tracking or updating internal state across signing operations. This simplifies its integration into embedded systems like a firmware TPM, where maintaining a persistent secure state can be both difficult and risky.

Another important advantage is the algorithm's clean and modular architecture. SPHINCS+ is built entirely on top of cryptographic hash functions, which not only avoids the need for complex mathematical structures but also makes it relatively easy to implement. In particular, updating or adapting the algorithm in the future is straightforward: it is sufficient to replace or modify the underlying hash functions, without requiring changes to the overall structure of the scheme.

This flexibility allows for easier adaptation to evolving standards or security requirements and ensures that SPHINCS+ remains a maintainable and future-proof solution. Furthermore, being entirely software-based, it does not rely on specific hardware features or accelerators, further simplifying deployment across different platforms and architectures.

2.7.3 Variants and Selection Criteria

SPHINCS+ offers different parameter sets that allow for trade-offs between performance and signature size. The two main variants are known as "s", the small variant, and "f", the fast variant. The "s" variant aims to reduce the size of the resulting signatures, while the "f" variant prioritizes speed in the signing process.

In this work, the "f" variant was chosen. This decision is based on the fact that signature generation is a critical operation in a firmware TPM, particularly in tasks such as attestation. While the "s" variant does offer some reduction in signature size, the difference is not significant enough to justify the performance overhead. In practice, the signatures produced by the "s" variant are still relatively large for a constrained environment, and the slower signing speed could impact the responsiveness of the system. The "f" variant, on the other hand, improves signing efficiency, which has a direct positive effect on the TPM's performance.

2.7.4 Strengths of SPHINCS-SHAKE-256f-simple

SPHINCS+ presents a series of advantages that make it an ideal candidate for secure post-quantum applications in embedded systems. First and foremost, its hash-based construction

relies on well-established and thoroughly studied primitives, offering strong theoretical security guarantees even in the face of future quantum adversaries.

The algorithm’s statelessness simplifies its integration into systems like a firmware TPM, where securely managing state over time is often impractical. The absence of a state not only reduces implementation complexity but also eliminates a common source of vulnerability in signature schemes.

Additionally, SPHINCS+ is entirely software-based, meaning it does not require dedicated cryptographic hardware or specific processor features. This increases the flexibility and portability of the implementation, allowing it to be deployed across a wide range of platforms with minimal adaptation effort.

Finally, the choice of the SHAKE-256f-simple variant offers a good balance between performance and practicality. The faster signature generation aligns well with the demands of TPM functionality, and the modular structure of the algorithm supports clean integration into the firmware’s existing architecture.

2.7.5 Proposed Solution: Integration into an fTPM

The proposed solution involved integrating SPHINCS+ directly into the firmware TPM (fTPM), to enable post-quantum digital signatures while maintaining full compatibility with existing TPM specifications. One of the guiding principles of this integration was to preserve adherence to the current Trusted Computing Group (TCG) standards so that the fTPM could remain interoperable with existing systems, software stacks, and protocols.

To achieve this, the algorithm was inserted into the TPM’s key management and signing components in a way that extends the original design, rather than replacing it. The result is a hybrid architecture where SPHINCS+ can coexist alongside traditional algorithms like RSA and can be activated through specific configuration parameters.

The integration was designed to be minimally invasive, modifying only the components necessary to support the generation and verification of post-quantum signatures. Internally, the system was adapted to handle SPHINCS+’s larger key and signature sizes, and appropriate serialization formats were introduced to ensure compliance with TPM data structures.

By embedding SPHINCS+ in this way, the implementation remains fully aligned with current TPM specifications, ensuring backward compatibility and reducing the risk of introducing inconsistencies at the interface level. This approach also facilitates the gradual adoption of post-quantum security features without disrupting existing workflows or infrastructure.

Chapter 3

Trusted Computing

This chapter provides an overview of the technological foundations, historical development, and key components of Trusted Computing. We will explore the evolution of Trusted Computing through the work of the Trusted Computing Group, understand the role and structure of the Root of Trust, and examine in depth the functionalities and types of the Trusted Platform Module (TPM). The goal of this chapter is to offer a comprehensive understanding of how Trusted Computing operates and why it is essential in the context of modern cybersecurity frameworks.

3.1 Introduction

3.1.1 Motivation and Background

In a world increasingly reliant on digital infrastructure, the need for secure and reliable computing systems is paramount. Trusted Computing (TC) was conceived to address this need by embedding trust mechanisms directly into computing platforms. It ensures security, privacy, and data protection through the integration of hardware-based security foundations [8].

3.1.2 Conceptual Framework

Trusted Computing is more than a set of technologies; it is a framework that redefines how we approach digital security. What sets it apart is its foundational reliance on hardware-level trust. Unlike traditional security models that rely heavily on software, Trusted Computing establishes a root of trust at the hardware level [32]. This means that the integrity and security of a system begin not with code that can be rewritten or compromised, but with physical components designed to be immutable and verifiable.

At the core of this architecture is the *Trusted Platform Module* (TPM), a dedicated microcontroller embedded in computing devices. The TPM provides a secure enclave for critical operations such as key generation, identity authentication, and platform integrity measurement. Because this trust anchor resides in hardware, it is significantly more resistant to tampering, malware, and unauthorized software modifications. By leveraging these hardware elements, Trusted Computing enables systems to measure, report, and attest their integrity, establishing a chain of trust from the firmware up through the operating system and applications.

3.1.3 Trusted vs Trustworthy

An essential distinction in this context is the difference between "trusted" and "trustworthy." A trusted component behaves as expected, though not necessarily secure; it simply means its behavior is known and predictable. Conversely, a trustworthy component guarantees it will not violate security policies, backed by formal verification or certification [33].

3.1.4 Relevance in Cybersecurity

TC is especially relevant in scenarios where attackers may gain full control over a system. In such cases, hardware-based protection is indispensable. The *Root of Trust* (RoT) [32], often realized via the TPM, serves as the initial anchor in the chain of trust, enabling mechanisms like secure boot, sealed storage, and remote attestation.

Later, this chapter will introduce what is the Trusted Computing Group, the entity that takes care of the trusted computing group, what are the cornerstones of trusted computing, and the state of the art of modern trusted computing technologies.

3.2 Trusted Computing Group

The formalization of Trusted Computing began with the U.S. Department of Defense's (DoD) 1983 *Trusted Computer System Evaluation Criteria* (TCSEC) [34], which introduced foundational concepts such as the Trusted Computing Base (TCB). This was followed by the Trusted Network Interpretation (TNI) and Trusted Database Interpretation (TDI), forming the earliest technical framework for TC.

In 1999, major industry players including IBM, HP, Intel, and Microsoft established the *Trusted Computing Platform Alliance* (TCPA). This initiative was rebranded in 2003 as the *Trusted Computing Group* (TCG) [35], marking a significant expansion in both scope and technical development. TCG produced specifications [36] for the *Trusted Platform Module* (TPM), *Trusted Software Stack* (TSS), *Trusted Network Connect* (TNC), and mobile platforms, which continue to evolve. In 2006, the European Open Trusted Computing (OpenTC) project [37] aimed to create open-source TC implementations. Comprising over ten academic and industrial partners, OpenTC tackled trusted computing through secure architectures for e-commerce, collaborative environments, and data centers. Today, TC is a standard feature in many commercial products. TPMs are embedded in almost all modern laptops and desktops. Technologies such as BitLocker, Windows Defender Credential Guard, and secure boot processes leverage TC to enforce platform integrity and protect user data.

3.3 Foundation of Trust: The Root of Trust

3.3.1 Definition and Purpose

Trust in computing must be rooted in a component that is verifiable, immutable, and resistant to tampering: the Root of Trust (RoT). To fully appreciate the significance of this concept, it is important to understand the notion of trust itself in the context of computing systems.

In everyday human interaction, trust is an abstract and often subjective notion, built on relationships, experience, and expectations. However, in computing, trust must be made concrete, measurable, and enforceable. Trust in this domain refers to the assumption that a given component will behave predictably and reliably, in alignment with defined policies and security objectives. In Trusted Computing, this trust must be rooted in hardware, as hardware is far less susceptible to manipulation than software.

The Root of Trust serves as the initial and most critical building block in any trusted computing architecture. Its integrity is essential, as every other trust decision within the platform is ultimately derived from it. If the RoT is compromised, all subsequent security operations, such as secure boot, measurement, and attestation, lose their foundation of reliability.

By anchoring trust in hardware, the RoT provides a hardened environment capable of performing security-critical functions without relying on potentially vulnerable software components. Its design prioritizes resistance to tampering, with carefully isolated execution paths and storage mechanisms. The presence of a reliable RoT ensures that trust in the system can be established

and extended upward through higher layers of the computing stack, from firmware to operating systems and application software.

In this sense, the Root of Trust is not just a technical feature but the fundamental enabler of all Trusted Computing guarantees. Without it, the ability to verify system integrity, protect cryptographic keys, and assure remote entities of a system's secure state would not be feasible. RoT is the anchor of all security functions in Trusted Computing. It provides mechanisms for integrity measurement, secure storage, and reporting.

3.3.2 Static vs Dynamic Trust

Generally, an entity is deemed trustworthy if its behavior aligns with expectations, both in the past and in the future. This concept extends to computing systems, where trust is typically divided into two categories: static and dynamic [8].

Static trust is established through a one-time evaluation based on predefined security criteria. The Common Criteria [38], an international standard for security evaluation, provides a structured framework for such assessments. These criteria define seven Evaluation Assurance Levels (EAL), with each higher level including the requirements of the levels below it. In static trust, a system's reliability is determined before its deployment and does not change over time.

Dynamic trust, on the other hand, adapts based on the system's real-time state and evolves through its operation. Here, the system's trustworthiness is continuously reassessed. If the calculated trust value deviates from the expected level, the system is no longer considered reliable. Dynamic trust relies on secure mechanisms that provide evidence of the system's current state. Trust in this context refers to the expectation that the system operates in a manner consistent with its security requirements. Dynamic trust often relies heavily on hardware components like the TPM, which can measure, store, and report the current system state. These real-time evaluations form the basis for decisions in secure boot processes, access control, and remote attestation protocols. While more complex to implement, dynamic trust provides a stronger and more flexible foundation for modern security architectures, especially in environments where system configurations and threats evolve frequently.

Together, static and dynamic trust models can complement each other, with static evaluations providing a certified baseline and dynamic mechanisms ensuring ongoing compliance and system integrity.

3.3.3 Types of Root of Trust

There are several types of Roots of Trust. The Root of Trust for Measurement (RTM) performs initial integrity measurements, typically via the Core Root of Trust for Measurement (CRTM). The Root of Trust for Storage (RTS) provides a secure storage area, only modifiable by trusted components. The Root of Trust for Reporting (RTR) ensures secure and verifiable communication of system states to external verifiers. These RoT elements interact to form a chain of trust where the RTM performs measurements, the RTS stores them securely, and the RTR reports them when requested.

3.4 Trusted Platform Module

3.4.1 Role in Trusted Computing

The Trusted Platform Module (TPM) is a critical cornerstone of trusted computing, implementing key RoT functions [32]. Essentially, the TPM is a cryptographic co-processor that can be either a physical hardware module or a virtualized entity. Its primary function is to secure sensitive operations, verify the integrity of systems, and enable trusted computing environments through

cryptographic mechanisms. By offering a root of trust, the TPM ensures that systems can be booted securely, secrets remain protected, and platform integrity is reliably measured and attested. It is a passive piece of hardware, meaning that software can interact with the TPM, but needs to do so explicitly. To give local or remote party guarantees, any software that runs on the target device needs to be measured successively by the TPM.

3.4.2 Key Security Features

The TPM incorporates advanced mechanisms to safeguard sensitive data, authenticate devices, and validate system integrity [39].

One of the core features of the TPM is its ability to generate and securely store cryptographic keys. These keys are held in an isolated environment, protecting them from system-wide threats. The TPM also supports device authentication using a unique Endorsement Key (EK), which is embedded during manufacturing. This key allows for the secure identification of devices.

The TPM measures the integrity of critical software components by calculating cryptographic hashes and storing them in Platform Configuration Registers (PCRs). These registers act as tamper-resistant logs, recording the system's state in a verifiable manner. The process of updating these registers is known as the extension operation. During this operation, when new data needs to be measured, the TPM does not overwrite the existing PCR value. Instead, it concatenates the current PCR value with the new hash (digest) of the data being measured and then applies a cryptographic hash function (typically SHA-1 or SHA-256) over the concatenated result. Mathematically, this is expressed as:

$$PCR_{new} = \text{hash}(PCR_{old} \parallel \text{digest_of_new_data})$$

This method ensures that the final value of a PCR is uniquely dependent on the entire sequence of measurements performed up to that point. As a result, even a single change in the order or content of the data being measured will lead to a different PCR value, making it infeasible for an attacker to recreate a legitimate system state by altering the measurement sequence.

This extension mechanism is fundamental for ensuring the integrity and authenticity of the system's state during operations such as secure boot and remote attestation. These registers act as tamper-resistant logs, recording the system's state in a verifiable manner. When new data is added, PCRs are extended using a cryptographic function that hashes the previous value with the new digest. This ensures that a given state can only be produced by a specific sequence of events.

Another critical feature of the TPM is sealed storage, which allows data to be encrypted in such a way that it can only be decrypted when the system is in a specific, trusted state. This functionality is further enhanced through the process known as secret sealing. With secret sealing, the data is bound not only to the TPM but also to a specific set of PCR values. This means that even if an attacker copies the sealed data to another system or attempts to change the system configuration, decryption will fail unless the platform is in the same state as when the data was originally sealed. This capability is essential for protecting sensitive information, such as cryptographic keys or credentials, in environments where system configurations might change or be exposed to tampering.

Through these mechanisms, the TPM forms the backbone of hardware-rooted trust in modern computing systems. to generate and securely store cryptographic keys. These keys are held in an isolated environment, protecting them from system-wide threats. The TPM also supports device authentication using a unique Endorsement Key (EK), which is embedded during manufacturing. This key allows for the secure identification of devices.

TPM serves as both the RTS and RTR. It functions as a secure storage unit and trusted entity for reporting, ensuring that only securely stored data is communicated. Additionally, through remote attestation, the TPM enables systems to prove their integrity to remote entities, fostering trust between different parties.

By integrating these capabilities, the TPM plays a vital role in trusted computing, ensuring the confidentiality of data, the integrity of the system, and secure device authentication.

3.4.3 Limitations of TPM

Despite its strengths, the TPM has limitations [32]. It is a passive component that only acts when called upon by the software. Its connection via the Low Pin Count (LPC) bus limits data throughput and introduces a potential attack vector. Specifically, an attacker with physical access to the system could tap into the LPC bus to intercept or manipulate communication between the TPM and the CPU [40].

In addition to LPC-related risks, the TPM offers only limited protection against physical attacks. Sophisticated adversaries may exploit physical tampering techniques or employ side-channel attacks to gain access to protected data. The TPM's reliance on asymmetric cryptography can be less efficient for some operations, and it was initially designed for PC platforms, limiting its adaptability to embedded systems, mobile environments, or high-performance servers without significant modifications. Lastly, the TPM was originally designed for PC platforms, which limits its adaptability for servers, embedded systems, and mobile computing environments without significant modifications.

3.4.4 TPM Versions

There are two TPM implementations that have defined the evolution of Trusted Computing: TPM 1.2 and TPM 2.0. The TPM 1.2 specification [41], released in 2011, laid the foundation for trusted platform security. It introduced critical features such as RSA cryptography with a minimum 2048-bit key size and a Random Number Generator (RNG) for generating secure randomness. The module also employed the SHA-1 algorithm for calculating integrity measurements. Although effective at the time, SHA-1 has since become cryptographically weak, necessitating improvements in future versions.

In TPM 1.2, each chip is provisioned with an Endorsement Key (EK), which serves as the immutable root of trust. Device manufacturers are expected to provide a certificate for the EK. The specification also introduced Attestation Identity Keys (AIKs) for digital signing and storage keys for securely encrypting and decrypting data. Platform Configuration Registers (PCRs) are included to store hash measurements of the system's state, a critical element for remote attestation.

Despite its foundational contributions, TPM 1.2 suffered from inconsistent implementations and reliance on aging cryptographic algorithms, driving the need for an updated architecture. An overview of the architecture of the TPM version 1.2 is shown in Figure 3.1.

Introduced in 2014, TPM 2.0 [42] addresses the limitations of its predecessor and offers significant enhancements. It includes support for modern cryptographic algorithms such as SHA-2 for hashing and Elliptic Curve Cryptography (ECC) for more efficient cryptographic operations. TPM 2.0 also increases the number of PCR banks, providing more flexibility in measurements and enhancing integrity reporting.

A notable improvement in TPM 2.0 is its hierarchical key management. Unlike TPM 1.2, which relied on a single key hierarchy, TPM 2.0 introduces separate hierarchies for Endorsement, Platform, and Storage keys. This structure enhances both security and manageability. Additionally, TPM 2.0 aligns closely with reference standards, improving consistency and interoperability across different devices and platforms.

Looking ahead, future TPM implementations will likely address quantum threats. Quantum-Resistant TPMs (QR TPMs) are expected to adopt co-processors specialized in lattice-based algebra. These would operate over polynomial rings defined by n dimensions modulus q , where

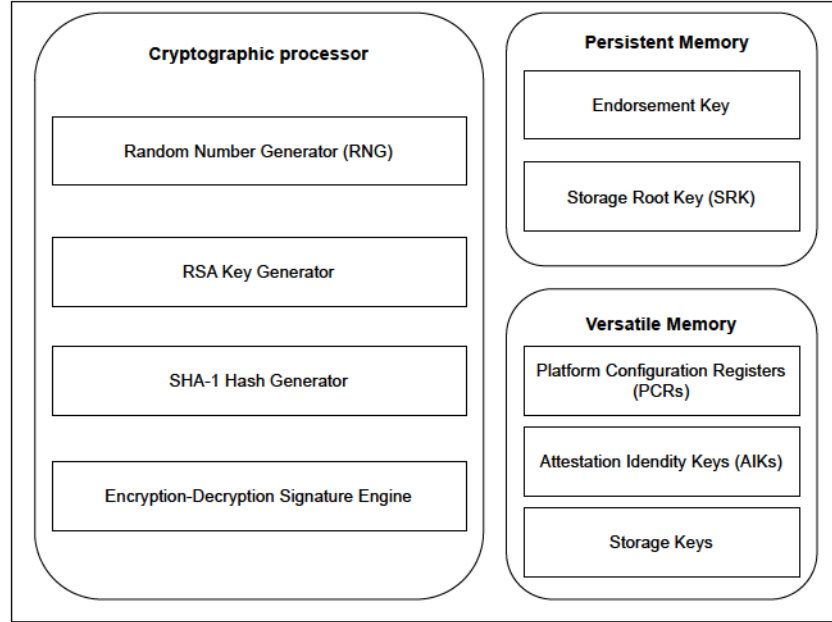


Figure 3.1: TPMv1.2 Architecture

q is a prime number. Operations in this model include addition, subtraction, multiplication, and division of polynomials, all performed modulo 1. QR TPMs will require a specialized vector engine in their cryptographic co-processor to efficiently handle such ring algebra.

To further optimize performance, QR TPMs will employ the Number Theoretic Transform (NTT) domain to reduce the complexity of algebraic operations. Research has shown that QR TPMs will require significantly larger buffers, with memory demands increasing by an order of magnitude due to the larger key sizes inherent in lattice-based cryptography. This evolution is anticipated to be reflected in future TPM architectures with an overall layout similar to current models but enhanced with expanded computational and storage capacity to cope with the increased size of post-quantum keys [3]. An example of future QR TPM is shown in Figure 3.2

3.4.5 TPM Implementations

TPM functionality can be implemented in various forms, each offering different trade-offs between performance, flexibility, and security. The most common types include hardware-based TPMs, software-based TPMs, and firmware-based TPMs.

Hardware-based TPMs

Hardware-based TPMs, also known as discrete TPMs (dTPM), are standalone chips integrated directly into a system's motherboard. These chips operate independently from the main CPU and system memory, offering a high degree of physical isolation. This makes them particularly resistant to software-level attacks and tampering. Their security benefits come with trade-offs, however. Because they often connect through the Low Pin Count (LPC) bus, they may suffer from limited data throughput. Furthermore, their static nature makes it difficult to update or patch them in response to new vulnerabilities, potentially requiring firmware updates or even hardware replacement. Physical attacks, such as side-channel or bus tapping, also pose a risk if an attacker has direct hardware access.

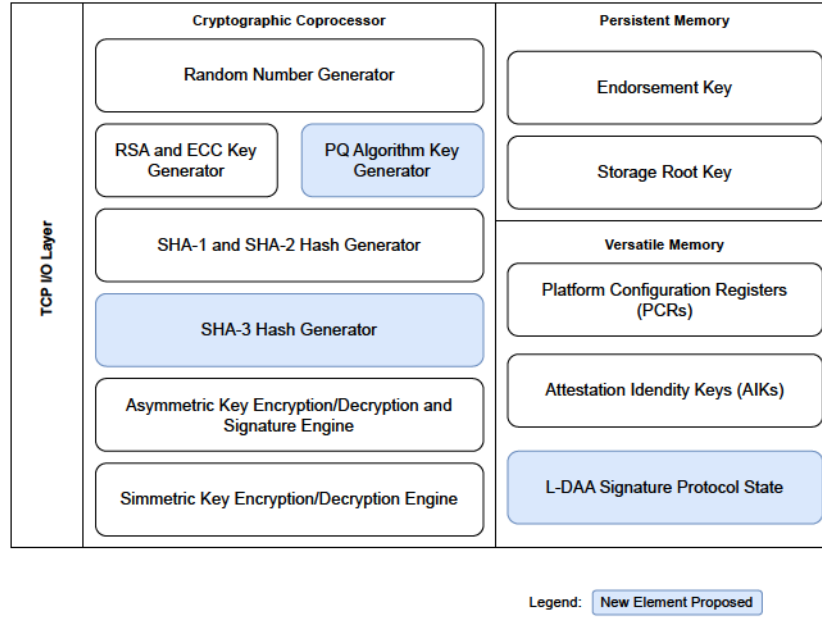


Figure 3.2: Architecture Proposed for QR TPM

Software-based TPM

In contrast to discrete TPMs, software-based TPMs emulate TPM functionality entirely through software. These implementations are commonly used in virtualized environments or for development purposes where physical TPM access is limited or unavailable. They offer greater flexibility and ease of deployment, particularly in systems that cannot accommodate additional hardware. However, this flexibility comes at the cost of security. Without the protection of physical isolation, software TPMs are vulnerable to a wide range of attacks, including memory snooping, privilege escalation, and exploitation of software bugs. They also face challenges in meeting compliance standards, especially for true random number generation, which in hardware TPMs relies on physical entropy sources.

Firmware-based TPM

Firmware-based TPMs (fTPMs) [43] represent a hybrid approach. They are implemented within the system's firmware and leverage the built-in security features of modern processors, such as ARM TrustZone or Intel SGX. These TPMs provide improved integration and performance over hardware-based solutions while maintaining a logical separation from the main operating system. Despite this, fTPMs are still susceptible to firmware vulnerabilities and do not offer the same level of isolation as discrete hardware TPMs. They are often used in consumer devices or cost-sensitive applications where full hardware TPMs are not practical.

3.5 Trusted Execution Environment (TEE)

3.5.1 Introduction

In recent years, the rapid expansion of sensitive applications across domains such as mobile banking, digital healthcare, cloud computing, and the Internet of Things (IoT) has brought increasing attention to the need for robust mechanisms to protect data and ensure secure execution. A key response to this demand is the Trusted Execution Environment (TEE), a secure and integrity-protected area of a processor that enables the isolated execution of code and management of

sensitive data [6].

Most modern processors integrate hardware-supported TEEs, which function in isolation from the standard operating environment, commonly referred to as the Rich Execution Environment (REE) [44]. The REE typically includes the device's operating system and user applications. The term "rich" refers to the extensive functionality offered by modern operating systems. However, this richness comes at the cost of a significantly larger and more vulnerable attack surface.

In contrast, the TEE operates as a lightweight and self-contained environment, dedicated to handling critical operations in a secure context. Its isolation ensures that, even if the REE is compromised, operations and data inside the TEE remain protected.

TEEs enhance both the security and usability of REE applications by restricting sensitive operations to the trusted environment. This guarantees that critical assets never leave the protected domain of the TEE. As a result, even when exposed to potentially untrusted software in the REE, sensitive computations remain safe. The TEE's security properties ensure confidentiality and integrity of all computations performed within its boundaries.

In addition, the TEE abstraction defines mechanisms for secure provisioning of both code and data into the environment. It also establishes trusted communication channels through which computational results and error messages can be securely retrieved by external entities, including REE applications or remote systems [45].

3.5.2 TEE Security Guarantees

The TEE provides strong guarantees of confidentiality, integrity, and authenticity for both code and data within its scope. These guarantees are made possible through a combination of hardware-enforced isolation and carefully designed software mechanisms. In addition to secure processing capabilities, a TEE also enables the secure provisioning of code and data, as well as trusted communication channels for reporting computation results and system errors.

A fundamental advantage of TEEs lies in their minimal Trusted Computing Base (TCB). Unlike systems that rely on large, complex secure subsystems or external secure elements, the TEE isolates only the essential components, significantly reducing the risk of security breaches. By acting as a trust anchor for the broader system [8], the TEE supports execution environments that demand high assurance without incurring unnecessary overhead.

3.5.3 Remote Attestation and Secure Storage

To establish trust with external systems, a TEE typically supports remote attestation. This process allows remote parties to verify the integrity and authenticity of the TEE before any sensitive data is exchanged or critical operations are requested. Remote attestation serves as a foundational step in building secure channels between the TEE and remote services, enabling protected and trusted communications even over untrusted networks [46, 47, 48].

In addition to these attestation capabilities, a TEE often has access to private secure storage, which is used to store sensitive application data such as cryptographic keys, certificates, or biometric templates. This storage is isolated from the rest of the system and is accessible only from within the TEE, thereby ensuring that the data remains protected even in the event of a compromise in the Rich Execution Environment [49].

TEEs enforce their security properties through a combination of hardware-based Root of Trust (HRT) and software-based mechanisms, creating a layered defense against both software and hardware threats. In some scenarios, TEEs are also used to implement the functionality of a Trusted Platform Module (TPM) entirely in software, effectively removing the need for additional specialized hardware [50]. This flexibility allows devices to support secure operations even when

dedicated security chips are not available, while still maintaining a high assurance level.

Together, secure storage and remote attestation enable the TEE to maintain a persistent, verifiable, and trusted state across multiple execution sessions and device reboots. These features are essential for the TEE's role as a long-term trust anchor in a wide range of security-sensitive applications.

3.5.4 Lifecycle Management and Hybrid Trust Model

One of the strengths of the TEE architecture is its ability to securely manage and update both code and data during the device's lifecycle. Unlike fixed secure elements or hardware coprocessors, the TEE is flexible and allows for the deployment of new trusted applications or updates as needed, without compromising its security guarantees.

The trust model adopted by TEEs is often referred to as hybrid trust [8]. This model incorporates both static trust, established during manufacturing or deployment, and semi-dynamic trust, which persists throughout runtime. During the initial boot process, the Root of Trust (RoT) verifies that only a certified TEE implementation is loaded onto the device. In ARM TrustZone-based systems, for instance, components like secureROM or eFuse serve as foundational trust anchors [51].

Once the TEE is running, its integrity is enforced and preserved by a separation kernel, which continues to isolate the TEE from the potentially compromised REE. This isolation ensures that the TEE's trustworthiness remains unchanged during operation, which is why it is considered semi-dynamic.

3.5.5 TEE Architecture

At the core of every Trusted Execution Environment (TEE) lies the architectural principle of *isolation*. The TEE operates as a secure and separate area within the main processor, running independently from the Rich Execution Environment (REE), where the standard operating system and user applications reside. This separation is enforced through a combination of hardware and software mechanisms, ensuring that sensitive operations and data remain protected even in the presence of a compromised REE.

The TEE hosts the execution of Trusted Applications (TAs), small, security-critical programs responsible for tasks such as cryptographic operations, biometric processing, or authentication. These applications are managed by a lightweight TEE runtime environment or Trusted OS, which controls memory allocation, access permissions, and execution flow.

Platform Integrity

Ensuring the integrity of the platform code is essential to maintaining the trustworthiness of the TEE. Integrity verification can take place both during the boot process and at runtime. During boot, two primary approaches are used: *secure boot* and *authenticated boot*.

In secure boot, the system verifies each stage of the boot sequence using code signing. The processor begins execution from an immutable memory region containing the initial bootloader, often stored in ROM. This bootloader verifies the next component in the chain using certificates and digital signatures, typically anchored to a public key provided by the manufacturer. If any verification fails, the boot process halts immediately. This approach ensures that only authenticated software is executed from the very first instruction. Cryptographic mechanisms used in this process can be secured by storing the required algorithms in read-only memory (ROM). The combination of an immutable boot sequence, a verification root, and protected cryptographic mechanisms provides the necessary trust anchors for a secure boot.

Authenticated boot, by contrast, involves measuring each component during startup, rather than verifying it directly. These measurements are stored in a tamper-evident log. They provide a verifiable record of the system state and can be used for local access control or remote attestation.

Since boot-time checks are not sufficient to prevent tampering at runtime, many systems implement runtime integrity verification. A trusted software or firmware component continuously monitors the state of platform code, and if any unauthorized modifications are detected, the system can restore original components or take corrective action [52, 53].

Secure Storage

Secure storage refers to a protected subsystem that allows sensitive data to be stored safely, even in the presence of a compromised REE. This storage mechanism typically relies on a device-specific hardware key, initialized during manufacturing and stored in a protected region of the chip. This key is accessible only by trusted code running within the TEE.

To defend against physical attacks, hardware-level protections such as coating, fuse-based keys, or tamper-evident packaging may be applied. Additionally, the secure storage system relies on trusted cryptographic primitives, including authenticated encryption and key derivation mechanisms.

To prevent rollback attacks, where an attacker attempts to restore an older, potentially vulnerable version of stored data, secure storage uses non-volatile writable memory, often with monotonic counters that persist across reboots and maintain the freshness of stored data.

Isolated Execution

The concept of isolated execution is fundamental to TEEs. It ensures that security-sensitive code can run independently of the REE, which is considered untrusted. When combined with secure storage, isolated execution enables applications such as credential handling, key management, and digital rights enforcement.

A TEE can securely load and execute custom trusted applications, which may implement either standard or proprietary cryptographic algorithms. These applications are granted controlled access to sensitive resources through the use of code certificates. These certificates also define the platform states in which the code is allowed to run, based on authenticated boot measurements.

A component known as the TEE management layer is responsible for enforcing access control policies and managing the lifecycle of trusted applications. Its integrity must be verified at boot time or dynamically during application loading, to ensure that only approved and untampered code is executed [54].

Attestation and Provisioning

Remote attestation is a security process that enables a device to produce an externally verifiable statement about its current software configuration. It allows service providers to determine whether a device complies with platform requirements before sharing secrets or granting access. This is typically achieved by signing cryptographic measurements of loaded firmware and software using a certified device key.

The secure delivery of secrets, credentials, or code into the TEE is known as provisioning. Establishing a secure provisioning channel requires that the device first authenticate itself using its unique key material. Since device certificates generally do not contain user identities, user authentication is handled through a separate mechanism.

Provisioned data is usually encrypted using the TEE’s certified key, ensuring that only the target TEE can decrypt and access the information. This process enables a secure, verifiable relationship between the service provider and the device, forming the basis for a wide range of trusted services.

Role of the Hardware Root of Trust

Underlying all the above components is the Hardware Root of Trust (HRT), a minimal set of hardware functions that establish the foundation for trust. The HRT is responsible for verifying the integrity of the TEE at boot and ensuring the authenticity of all code and data loaded into it.

TEEs enforce their security properties by leveraging this HRT in conjunction with their software stack. In some implementations, TEEs are also used to emulate the functionality of a Trusted Platform Module (TPM) entirely in software, eliminating the need for separate secure elements while still delivering a comparable level of assurance [50].

3.5.6 Applications

Several notable implementations of the TEE concept exist today. Intel Software Guard Extensions (SGX) and ARM TrustZone are two of the most widely used platforms, each offering different architectural approaches and use cases. Other examples include AMD SEV (Secure Encrypted Virtualization) and Google’s Titan M chip.

The next chapters will provide a detailed overview of ARM TrustZone, explaining its architecture, operational model, and security mechanisms. This will be followed by a comparative analysis with Intel SGX, highlighting the key differences in their design choices, isolation strategies, and real-world use cases. This comparison will serve to illustrate the trade-offs between flexibility, performance, and security in contemporary TEE implementations.

3.6 ARM TrustZone

3.6.1 Introduction

ARM TrustZone, introduced in [55] and [56], is a hardware-based security extension designed to support secure execution environments within a single system-on-chip. It is widely adopted in mobile, embedded, and IoT platforms to isolate sensitive code and data from the rest of the system. TrustZone introduces a dual-world model, dividing execution into the Secure World, which handles security-critical tasks, and the Normal World, which runs the main operating system and general-purpose applications.

These two worlds coexist on the same processor but are strictly separated through hardware-enforced mechanisms. This separation is orthogonal to the traditional ARM privilege model and operates across all exception levels. As a result, each privilege level (EL0 to EL3) exists in both secure and non-secure contexts.

3.6.2 Processor Mode Separation

The separation between the two worlds is enforced through dedicated hardware mechanisms. Software running in the Normal World is entirely barred from accessing resources belonging to the Secure World. One of the key architectural elements involved in this separation is the System Control Coprocessor, known as CP15. CP15 is a privileged coprocessor present in ARM processors that manages system-level operations such as virtual memory translation, cache behavior, exception handling, and other low-level system controls.

In TrustZone-enabled systems, CP15 registers are logically duplicated to maintain independent configurations for the Secure and Normal Worlds. For instance, memory management units (MMUs) and translation look-aside buffers (TLBs) have separate secure and non-secure states, ensuring that sensitive memory regions configured by the Secure World cannot be altered or accessed by software in the Normal World. Additionally, critical processor status bits are either hidden from the Normal World or have access permissions strictly governed by the Secure World.

3.6.3 Exception Levels and Execution Contexts

In addition to world separation, ARM introduced the concept of Exception Levels (EL), which provide a hierarchical model of privilege. These levels, ranging from EL0 to EL3, determine access rights to hardware and system resources [57]. EL0 is typically used for user-space applications, EL1 for the operating system kernel, and EL2 for hypervisors in virtualized environments. The highest privilege level, EL3, is exclusively reserved for trusted firmware operating in the Secure World. This layered approach allows fine-grained control over access to the system and plays a critical role in enforcing the platform's security policies.

The EL model provides the foundation for the separation of roles and isolation, not just between the operating system and applications, but also across secure and non-secure domains. This model allows TrustZone to assign specific exception levels to each world, thereby tightly controlling which components can perform critical operations such as context switching, memory management, or interrupt handling.

In TrustZone-enabled systems, both the Secure and Normal Worlds have their own stack of exception levels. At the top of this hierarchy, EL3 operates only in the Secure World and hosts the Secure Monitor, which acts as a gatekeeper between the two worlds.

This layered model not only reinforces privilege boundaries within a world but also across worlds, forming the architectural basis for TrustZone's world isolation and secure control flow. A diagram summarizing the relationship between Exception Levels and TrustZone's execution environments is provided in Figure 3.3.

	Normal World	Secure World
EL0	Normal Application	Trusted Application
EL1	Normal OS	TrustZone OS (OP-TEE)
EL2	Hypervisor	
EL3	ARM TrustZone Firmware	

Figure 3.3: ARM TrustZone Privileges Architecture

3.6.4 World Identification and the NS Bit

The processor identifies its current execution world using a dedicated signal known as the *Non-Secure* (NS) bit. This is effectively the 33rd bit in the memory addressing scheme and acts as a flag to distinguish between Secure and Normal World execution contexts. Hardware components and system software use this bit to enforce security boundaries, ensuring that access permissions and execution rights are applied consistently across the platform.

3.6.5 Memory and Peripheral Isolation

TrustZone systems include dedicated hardware to support memory and device partitioning. Notably, the *TrustZone Address Space Controller* (TZASC) and the *TrustZone Memory Adapter* (TZMA) provide low-level mechanisms for isolating memory access between worlds [58, 59].

The TZASC enables the partitioning of DRAM into secure and non-secure regions. These memory regions are configured via control registers that are only accessible from the Secure World. Applications running in the Secure World may access memory assigned to the Normal World, but the reverse is strictly prohibited. This guarantees unidirectional access and prevents unauthorized reads or writes from non-secure software.

The TZMA performs a similar role of TZASC but for off-chip ROM and SRAM. It also supports the configuration of coprocessors that extend core instructions or register sets. These extended instructions and registers are specific to their respective worlds and can only be accessed or modified within their corresponding domain.

To complement memory isolation, TrustZone also enables protection at the peripheral level. Using optional components such as the *TrustZone Protection Controller* (TZPC), devices can be assigned to either the Secure or Normal World. Any access attempt from the wrong domain triggers hardware exceptions or is silently blocked. This mechanism protects critical resources such as secure key storage, interrupt controllers, and debug interfaces from being tampered with by non-secure software.

3.6.6 Virtual Memory

The *Memory Management Unit* (MMU) in ARM TrustZone is aware of TrustZone's execution context and maintains separate translation tables for each world. It distinguishes between secure and non-secure entries using the *Non-Secure TLB ID* (NSTID), an additional bit used within the *Translation Lookaside Buffer* (TLB) [60].

This enables each world to define its own virtual-to-physical memory mappings, ensuring complete isolation in address translation. Secure applications, known as Trusted Applications (TAs), are generally confined to a small amount of on-chip memory. Due to the high cost and limited capacity of secure memory, these applications are intentionally designed with minimal memory footprints, including only the functionality strictly necessary for their operation. This constraint not only optimizes performance but also reduces the attack surface, enhancing overall system security.

3.6.7 Secure Boot and Chain of Trust

TrustZone plays a fundamental role in establishing a secure boot process, which is essential to guarantee the integrity and authenticity of all software components from the very first instruction executed. To ensure the overall security of the system, protection must be enforced from the very beginning of the boot sequence.

When the device is powered on and reset, the processor begins execution in the Secure World. A first-stage bootloader, typically stored in ROM, is automatically loaded and executed. This initial firmware is implicitly trusted since it resides in read-only memory and cannot be modified. It is responsible for initializing critical peripherals and performing an integrity check on the second-stage bootloader, which is usually stored in non-volatile memory such as flash memory.

If the verification of the second-stage bootloader succeeds, control is passed to it. The second-stage bootloader then performs a similar validation step on the Secure World operating system, ensuring that only authenticated and untampered code is loaded. Once the secure OS is verified and launched, the Normal World operating system is started. Some secure OS implementations

further extend the chain of trust by verifying the integrity of Trusted Applications (TAs) before allowing them to execute.

The entire process forms a chain of trust, where each layer verifies the next before transferring control. TrustZone relies on standard public-key cryptography to perform these integrity checks. Code images are signed by the vendor using a private key, and the firmware uses a corresponding public key to verify these signatures at runtime. To support multiple vendors and enable flexibility in deployment, the architecture allows for the storage and use of multiple public keys, enabling devices to validate software signed by different trusted authorities.

Through this mechanism, TrustZone ensures that all code executed in the Secure World is authenticated and trusted, effectively preventing the execution of malicious or unauthorized software during boot.

3.6.8 Secure Storage and Replay Protection

TrustZone supports persistent data storage for Trusted Applications (TAs) through a secure storage mechanism. This storage is designed to ensure the confidentiality, integrity, and authenticity of sensitive data, even when stored on potentially untrusted media.

Objects stored by TAs are encrypted on disk and digitally signed as an anti-tampering measure. Although the data resides in encrypted form externally, TAs access it in cleartext, as the TEE layer handles all cryptographic operations transparently. This abstraction allows developers to store sensitive data securely without manually managing encryption or signatures.

Each file stored in secure storage is assigned a unique numeric identifier, typically derived from a monotonic counter. An encrypted index of all stored objects is maintained in parallel, which maps these identifiers to their respective locations. To protect the consistency and integrity of this index, TrustZone uses a hash tree data structure, which allows file system operations such as reads, writes, and deletions to be performed atomically and verifiably.

To defend against storage replay attacks, in which an attacker attempts to roll back secure storage to a previously valid state, TrustZone systems leverage features of modern storage hardware. Specifically, secure storage is backed by eMMC devices (embedded MultiMediaCards), which are non-volatile, soldered memory modules commonly found in mobile and embedded systems [61]. These devices include a special secure region known as the Replay Protected Memory Block (RPMB) [62]. The RPMB allows authenticated, write-protected access to monotonic counters and protected data, ensuring that old or replayed data cannot be injected after reboot.

3.6.9 World Switching and Secure Monitor

Switching between the Secure and Normal Worlds is a key operational mechanism in TrustZone. This functionality is handled through a dedicated privilege level, Exception Level 3 (EL3), where the processor operates in Secure Monitor Mode. This mode is responsible for managing world transitions, ensuring the integrity and isolation of the two execution domains.

TrustZone introduces a specialized processor mode specifically designed to switch between the two worlds by saving and restoring the processor's full state [63]. When a transition occurs, the Secure Monitor saves the current context (including registers and processor status) upon entering the Secure World and restores it upon returning to the Normal World. This operation is tightly coupled to the Non-Secure (NS) bit, a critical flag that indicates the current security domain of execution: a value of $NS = 1$ denotes the Normal World, while $NS = 0$ corresponds to the Secure World. The Secure Monitor toggles the NS bit during the transition, which is managed internally by the monitor mode logic.

At privilege level EL1 or PL-1 in ARMv7 terminology [64], transitions are triggered by a dedicated hardware instruction known as Secure Monitor Call (SMC). When software running in the Normal World requires access to a trusted service, such as a cryptographic operation handled by a Trusted Application (TA), it issues an SMC instruction. This instruction signals the CPU to pause the Normal World's workflow and transfer control to the Secure World, where the requested task is executed. Upon completion, another SMC call is issued to return control to the Normal World and resume its operation flow.

Modern ARM processors, such as those based on the Cortex-A family [65], support this mechanism natively and allow the operating system kernel to issue SMCs as part of inter-world communication. Importantly, world switching is core-specific, meaning that each CPU core can only execute instructions in one world at a time. Consequently, the number of available cores may limit the degree of parallelism when interacting with TAs from multiple threads.

A kernel thread in the Normal World typically initiates a transition by invoking the Secure Monitor, which then carries out the SMC. If an SMC instruction is issued by a process not operating in privileged mode, the processor triggers an undefined instruction exception [66], preventing unprivileged software from accessing Secure World services directly.

From a software architecture perspective, applications in the Normal World access TrustZone functionalities indirectly, by calling APIs exposed by the REE kernel or middleware layers, which in turn issue SMCs to the Secure World. Trusted Applications may also call one another internally within the Secure World, a capability that enables modular design and helps minimize code duplication and reduce the attack surface.

Communication between worlds typically occurs through shared memory pointers or via direct memory copies. This approach provides efficiency, while the Secure Monitor and TEE enforce strict validation and memory access policies to prevent data leakage or tampering across domains.

3.6.10 Interrupts in ARM TrustZone

Interrupt handling is a critical component of the TEE as it helps protect the system from malicious software attempting to exploit interrupt vectors. The TrustZone architecture enhances the Generic Interrupt Controller (GIC) by supporting prioritized secure and non-secure interrupt sources. This prioritization is essential to mitigate denial-of-service (DoS) attacks from non-secure software, as secure interrupts can take precedence over non-secure ones. Depending on the GIC configuration, various interrupt models can be implemented for handling normal interrupts (IRQs) and fast interrupts (FIQs). The FIQ mode is reserved for devices allocated to the secure world's memory region, ensuring that lower-priority IRQs cannot interrupt secure world execution. If an interrupt occurs in its respective environment, no context switch is required. However, when an IRQ triggers while in the TEE, the monitor must switch to the appropriate execution environment to address the interrupt.

TrustZone employs the CP15 coprocessor [67] to safeguard interrupt management. The CP15 includes a control register accessible only in the secure world, preventing software in the REE from altering the F and A bits in the Current Program Status Register (CPSR). The F bit masks FIQ interrupts, while the A bit masks external interrupts. This mechanism ensures that malware in the REE cannot block interrupts destined for the TEE.

Separate exception vector tables are used to define interrupt service routine addresses for the secure world, normal world, and monitor mode. Each mode has its own vector table base address, which can only be updated by the corresponding mode, ensuring that secure interrupt sources remain unaffected by normal-world software manipulation.

The interrupt management features of TrustZone-enabled ARM cores provide flexibility, allowing for various approaches to handling secure and non-secure interrupts. As detailed in [68], two key properties of IRQ and FIQ interrupts can be configured by secure-world privileged code:

- Non-secure world access to the global interrupt disable bit for FIQ interrupts can be restricted, preventing unauthorized modifications
- IRQ and FIQ interrupt destinations can be directed either to the regular vector tables of the current world or to the Secure Monitor Mode vector table

A recommended configuration, as highlighted in [68], involves disallowing the normal world from modifying the FIQ disable bit and using a Secure Monitor Mode FIQ handler. This setup enables deterministic secure interrupts. A straightforward interrupt strategy could assign IRQs exclusively to the normal world and FIQs to the secure world. This approach works well in systems with a single secure world OS and a single normal world OS.

To prevent interference between non-secure world compartments, the secure world kernel must have complete control over interrupt handling. Non-secure compartments must not be allowed to mask or disable interrupts assigned to other compartments running concurrently.

3.6.11 Key Management in ARM TrustZone

The key manager starts with a device-specific key, the Secure Storage Key (SSK). It is derived from two pieces of information unique to each device's processor: the chip identifier and the hardware key. The TA Storage Key (TSK) is a per-TA key, derived from the SSK and the TA's UUID identifier. The File Encryption Key (FEK) is a per-file key generated upon file creation. It is used to protect the file contents, including its metadata, and is encrypted using the TSK.

3.6.12 Shortcomings of ARM TrustZone

Although the ARM TrustZone specification describes how the processor and memory subsystem are protected in the secure world and provides mechanisms for securing I/O devices, the specification is silent on how many other resources should be protected. This has led to a fragmentation of implementations, which at the same time has its pros and cons. The observations below held across all the major SoCs vendors when products based on this work shipped.

No Trusted Storage

Surprisingly, the ARM TrustZone specification offers no guidelines on how to implement secure storage for TrustZone. The lack of secure storage drastically reduces the effectiveness of TrustZone as trusted computing hardware. Naively, one might think that code in TrustZone could encrypt its persistent state and store it on untrusted storage. However, encryption alone is not sufficient because one needs a way to store the encryption keys securely, and encryption cannot prevent rollback attacks [69].

Lack of Secure Entropy and Persistent Counters

Most trusted systems make use of cryptography. However, the TrustZone specification is silent on offering a secure entropy source or a monotonically increasing persistent counter. As a result, most SoCs lack an entropy pool that can only be read from the secure world, and a counter that can persist across reboots and cannot be incremented by the normal world [70].

Limitation on the memory isolation

Although TrustZone offers a separation between the secure world and the normal world, some vulnerabilities can compromise this separation. ARM TrustZone is based on low-level language, Assembly, and C. In these languages, developers have to manage every allocated memory checking. One of the major limitations in the ARM TrustZone framework is, it does not have any in-built memory management support, even for secure zones. This opens the door for the overflow of the memory in a secure zone and possible leakage of valuable data.

Lack of access

Most SoC hardware vendors do not provide access to their firmware. As a result, many developers and researchers are unable to find ways to deploy their systems or prototypes to TrustZone. SoC vendors are reluctant to give access to their firmware [43]. They argue that their platforms should be "locked down" to reduce the likelihood of "hard-to-remove" rootkits. Informally, SoC vendors also perceive firmware access as a threat to their competitiveness. They often incorporate proprietary algorithms and code into their firmware that takes advantage of the vendor-specific features offered by the SoC. Opening firmware to third parties could expose more details about these features to their competitors.

3.6.13 Comparison between technologies

ARM TrustZone and Intel SGX are mainstream TEE technologies that aim to create a secure and isolated environment for sensitive task computing and private data storage to prevent attackers from obtaining data and harming system security. They are used on different platforms and application scenarios, so they adopt other design concepts, making a massive difference. This section analyses the security protection of ARM TrustZone and SGX from the perspectives of their design concepts, isolation protection principles and operation mechanism. In Table 3.1 are summarized the differences between ARM TrustZone and Intel SGX.

Design Concept

ARM TrustZone and SGX both guarantee a trusted execution environment at runtime, so that malicious code cannot access and tamper with the protected content of other programs at runtime, enhancing the security of the system, but they adopt different design concepts. The design concept of TrustZone is based on the CS model design, which constructs two separate worlds, set the Trusted Execution environment for the secure world. The SGX is based on the P2P model design. The CPU in TrustZone works in the secure world and the normal world, and the two worlds communicate with each other through SMC instructions. But in SGX, a CPU can run multiple secure enclaves and can run parallelly, and the memory occupied by the enclave will encrypt the hardware.

Trusted Basis Design

The TrustZone uses the entire secure world as the trust base, including security components, secure operating systems, and secure applications. Normal world applications share the same trust base. The secure world is configured by device manufacturers, thereby simplifying user development and use. However, there is a lack of effective isolation between secure applications in the secure world. The failure of any secure application will lead to the loss of the entire trust base. SGX regards enclave as an independent, trusted base, which corresponds to applications one by one. Enclave does not pose a threat to the system and other enclave security but increases the difficulty of user development and maintenance.

Security Service Design

TrustZone deploys vendor-designed secure code in the secure world ahead of time, and normal world applications can only request fixed generic security services through vendor-provided security interfaces. TrustZone can not provide dedicated services for the normal world beyond secure code. SGX security service is more specialized. SGX applications are divided into secure part and nonsecure part, which are developed by users. The secure part is deployed in the enclave and runs in isolation, creating different secure codes according to different functions.

Task Scheduling

TrustZone and SGX are designed for multi-core systems and support virtual machines. The TrustZone processor makes only one secure call at a time per core, supported by the Monitor module for world state switching. SGX endorses the execution of multiple enclave threads, and the running process processor can respond to interrupt execution, so the task scheduling is more flexible than TrustZone.

Isolation Protection Principle

ARM TrustZone and SGX adopt different Isolation protection principles. TrustZone provides a secure world that operates independently of the host which contains all secure operations. Since TrustZone is only divided into secure world domain and normal world domain, TrustZone only needs to formulate isolation protection policies around the secure world, which uses software and hardware to divide resources between the secure world and the normal world. In software, a dedicated operating system in the secure world is a complex, but powerful, design. It can simulate concurrent execution of multiple independent secure world applications, runtime download of new security applications, and secure world tasks that are completely independent of the normal world environment. Secure bits are extended on hardware to isolate resources such as memory and I/O using auxiliary controllers such as Advanced eXtensible Interface AXI, TZASC, TZMA, and TrustZone Protection Controller to provide hardware resources required for the operation of the secure world. The Monitor is a security critical component, as it provides the interface between the two worlds. The Monitor module runs in the highest privileged state. The Monitor module runs in the highest privileged state. It connects the normal world and the secure world, isolates access between different worlds, and provides system-level secure protection.

SGX allows users to actively create and maintain enclaves, deploy and apply secure code and private data, and provide application-level protection. Each enclave acts as an independent secure environment. Unlike TrustZone, SGX needs to protect different enclaves, so SGX adopts different isolation protection methods. SGX offers a set of instructions that applications can use to create a private region of memory that is isolated from all other processes, even those with higher privilege levels. Thus, even if a malware or an insider has access to operating system (OS) root privileges, or if the virtual machine manager (VMM) or BIOS are compromised, the SGX-protected application can still operate with integrity and be able to help protect both its code and data.

Operating Mechanism

In ARMTrustZone, the normal world and secure world are two independent environments. A secure service request in normal world contains two procedures:

- World state switch, which includes switching from the normal world to the secure world and from the secure world to the normal world.
- Execute secure operations: the Monitor module switches to the secure world when the normal world sends a secure service request. After that, the secure world responds to the normal world request, executes secure operations, and returns the result to the normal world.

In Intel SGX, users create enclaves and deploy secure codes and private data in the enclave. SGX protects them from being accessed by external software. Enclave can prove its identity to remote authenticators and provide the necessary functional structure for securely providing keys. Users can also request a unique key, which is unique by combining the enclave's identity with the platform's identity, and can be used to protect keys or data stored outside the enclave. The application requests the secure enclave operation when executing, which needs to set the processor to the enclave model. The processor executes the secure operations and returns the processing results. Compared to TrustZone, SGX executes enclave switching with fewer costs. Although frequent encryption and decryption steps are involved in SGX, the complete hardware design reduces the encryption and decryption time. However, besides normal, secure operations,

Trust-Zone also executes world status switching and save a large amount of context information, which results in a long time for a single secure request. In [10] was demonstrated how the single switching cost of the TrustZone world state is about seven times of the single switching cost of the SGX enclave.

Feature	ARM TrustZone	Intel SGX
Design Concept	Divides the system into two worlds: normal world and secure world, communicating via SMC	Creates independent enclaves that isolate sensitive code and data through hardware encryption
Trust Base	The entire secure world is considered the trust base. A failure compromises the whole system	Each enclave is a separate trust base. A compromise does not affect other enclaves
Security Service	Generic services predefined by device manufacturers; limited flexibility	Custom services: secure code is developed by users and isolated within enclaves
Task Scheduling	Supports only one secure service per core at a time. World state switches (normal/secure) are slow	Supports parallel execution of multiple threads within enclaves. More efficient task scheduling
Isolation Principle	Isolation between normal and secure worlds via dedicated software and hardware	Application-level isolation with private memory protected even from privileged processes
Performance	World state switches between normal and secure worlds are slow and time-consuming	Enclave state switches are faster due to advanced hardware design
Development Flexibility	Simple for users: configured by device manufacturers	More complex: users must create and manage enclaves
Protection	Limited protection: the secure world is vulnerable if the OS or secure applications fail	Strong protection: enclaves are independent and isolated even against compromised OS or insiders

Table 3.1: Differences between ARM TrustZone and Intel SGX

3.7 Remote Attestation

3.7.1 Introduction

Remote Attestation is the process by which an authorized party verifies that a particular platform is in a trustworthy and unaltered state. This mechanism plays a fundamental role in building trust in distributed environments, especially when direct control over a system cannot be assumed. Trusted computing architectures, such as those involving a Trusted Platform Module (TPM), are designed to support both local and remote attestation. While local attestation occurs between software modules within the same platform, remote attestation involves a verifier that resides outside the system, typically communicating over a network.

Remote Attestation is particularly important when assessing whether a platform has been compromised. Due to the inherent limitations of self-assessment, the TPM cannot be solely relied upon to evaluate its own platform; instead, it must work in conjunction with an external verifier. This external party is responsible for evaluating the integrity of the system based on cryptographic evidence provided by the TPM.

3.7.2 Attestation Workflow

The process involves two main actors: the attesting platform, equipped with a TPM, and a remote verifier. When the verifier wishes to assess the platform's trustworthiness, it sends a unique

challenge to the platform. The TPM responds by collecting the current values stored in its Platform Configuration Registers (PCRs), which are cryptographic hashes representing the integrity of various software and hardware components. These values are then digitally signed using a device-specific key, often referred to as the Device Identity Key (DevID). Since the response is bound to the received challenge, replay attacks are prevented: the signed evidence is valid only for that specific request.

The remote verifier performs two verification steps. First, it checks the digital signature to ensure the integrity and authenticity of the evidence and to confirm that it was produced by the correct TPM. This is done by identifying the TPM's public key and matching it with the signed response. Second, it compares the PCR values with a set of pre-established reference measurements, also known as *golden values*. If the measurements match the expected values, the platform can be considered trustworthy.

Importantly, these measurements depend not only on the software present on the system, but also on the order in which components were loaded. This sequence sensitivity adds an additional layer of protection, making it more difficult for an attacker to replicate a trusted state through simple substitution.

3.7.3 PCR Usage in PC Clients

According to the specifications provided by the Trusted Computing Group (TCG), PCRs are assigned specific roles in measuring platform integrity. For instance, PCR0 stores a hash of the platform firmware located in the motherboard's ROM and is expected to remain constant for a given firmware version. PCR1 extends this measurement to include additional firmware components, while PCR2 records the hashes of drivers loaded from disk. PCR4 reflects components related to the UEFI or legacy OS loader, and PCR5 captures aspects of the underlying platform hardware. PCR7, on the other hand, contains the Secure Boot policy enforced by the platform.

A further explanation of the content of those PCRs is shown in Table 3.2. Registers from PCR8 onwards are typically left to the operating system, which may use them to monitor dynamic aspects of the system. However, the values stored in PCRs 0 through 7 are considered predictable and verifiable. Their expected states are part of the golden measurements, and any deviation from these values indicates a potential compromise. Thus, if incorrect or unexpected values are found in these registers during attestation, the system cannot be considered trusted.

PCR Index	PCR Usage
0	SRTM, BIOS, Host Platform Extensions, Embedded Option ROMs and Pi Drivers
1	Host Platform Configuration
2	UEFI Driver and Application Code
3	UEFI Driver and Application Configuration and Data
4	UEFI Boot Manager Code and Boot Attempts
5	Boot Manager Code Configuration and Data and GPT/Partition Table
6	Host Platform Manufacturer Specific
7	Secure Boot Policy
8-15	Defined for use by the static OS
16	Debug
23	Application Support

Table 3.2: PCRs Usage

Chapter 4

Technologies Used

This chapter presents the technologies used in the development of the project, outlining their key features and explaining the rationale behind their selection of the established objectives.

4.1 OP-TEE

When developing applications for TEEs, available options are limited. However, OP-TEE stands out as an ideal choice due to its fast development cycle and native support for ARM TrustZone. This technology allows the creation of a secure execution environment, protecting sensitive data and critical operations from potential threats.

OP-TEE is an open-source project that implements TrustZone technology. It is managed and distributed by Linaro, a nonprofit organization dedicated to supporting open-source software on ARM platforms. Thanks to its flexibility, OP-TEE can be used alongside a Linux-based operating system running in the REE, providing a balance between security and usability.

4.1.1 OP-TEE Architecture

OP-TEE is designed to offer a secure execution environment composed of several key components that work together to ensure reliable execution of Trusted Applications (TAs).

At the core of this system is Op-tee Os [71], a minimal operating system running in the Secure World. Supporting this architecture is the tee-supplciant [72], a service operating in the Normal World, which helps the TEE access resources from the main operating system.

To facilitate development it provides a complete toolchain [73] for building and debugging TAs, along with a testing suite [74] to verify system functionality. Additionally, it includes a set of utilities that simplify security management and integration with different hardware platforms.

OP-TEE is highly adaptable and can be deployed on ARM platforms where the manifest file specifies the necessary dependencies and hardware characteristics. Developers can also use the QEMU emulator [75] to run OP-TEE in a virtual environment, allowing for testing and evaluation without requiring ARM hardware.

Another crucial aspect of OP-TEE is its compliance with GlobalPlatform specifications, ensuring compatibility with industry security standards.

4.1.2 Communication Between REE and TEE in OP-TEE

Since the Normal World (REE) and Secure World (TEE) are isolated, they cannot communicate directly. Instead, OP-TEE defines a structured interface for Client Applications (CAs) running

in the REE to interact with Trusted Applications (TAs) in the TEE.

How Communication Works

When a Client Application (CA) in the Normal World requires a secure operation, it sends a request to OP-TEE using the TEE Client API. This request must include a Universally Unique Identifier (UUID), which serves as a unique identifier for the requested Trusted Application (TA). Every TA must have a UUID defined at compile time, ensuring that it remains uniquely identifiable across different applications. Figure 4.1 illustrates the interaction between the CA and TA. Once

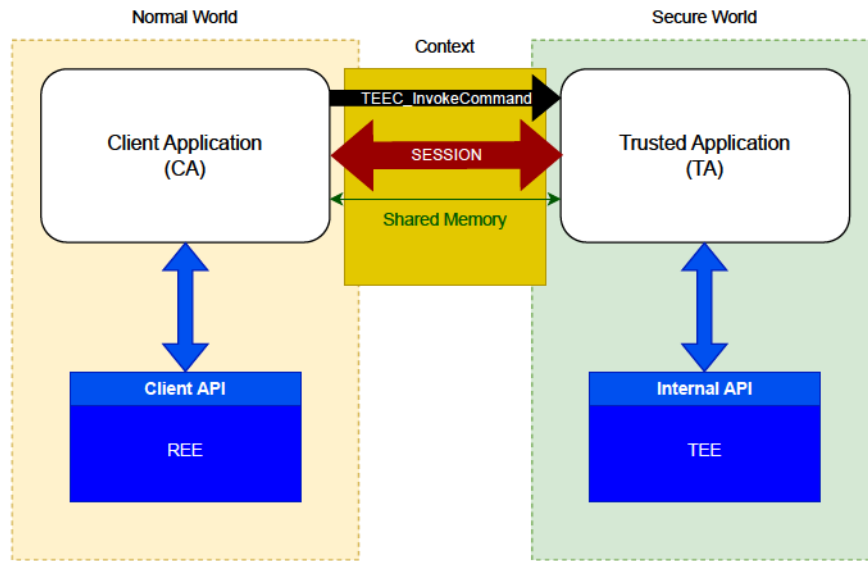


Figure 4.1: TEE CA and TA Communication

the UUID is provided, OP-TEE initializes a context via the function *TEEC_InitializeContext*, which references the TEE driver. With this context, the CA then invokes *TEEC_OpenSession* to establish a session with the TA. At this stage, OP-TEE loads the TA from the Normal World into the Secure World with the help of the tee-supplciant, which ensures that the requested TA is properly initialized and ready for execution.

Invoking Commands in a TA

Once a session is established, the CA can invoke services within the TA using the function *TEEC_InvokeCommand*. This function allows the CA to execute specific commands within the TA while passing parameters that contain either values or references to shared memory.

An important feature of OP-TEE is that within a single session, the *TEEC_InvokeCommand* function can be called an unlimited number of times, enabling flexible and efficient execution of multiple commands without requiring the session to be reopened. This design significantly improves performance by reducing session-handling overhead while maintaining a secure communication channel between the CA and TA.

When the execution is complete, the session can be closed using *TEEC_CloseSession*, and the context is released by calling *TEEC_FinalizeContext*, ensuring proper resource management. Table 4.1 summarizes the client-side functions used to interact with and establish a connection to a Trusted Application (TA).

Function	Description
<i>TEEC_InitilizeContext</i>	Create a context, a logical connection between TEE and CA
<i>TEEC_FinalizeContext</i>	Release the logical connection stored in the context
<i>TEEC_OpenSession</i>	Create session by connecting TA and CA specified by UUID
<i>TEEC_InvokeCommand</i>	Service request by function or service ID of TA connected to Session
<i>TEEC_CloseSession</i>	Terminate TA connection with CA stored in Session
<i>TEEC_RegisterSharedMemory</i>	Register CA's memory block in shared memory in context scope
<i>TEEC_AllocateSharedMemory</i>	Allocate CA memory block to shared memory in context scope
<i>TEEC_ReleaseSharedMemory</i>	Deallocate the block of memory from shared memory
<i>TEEC_RequestCancellation</i>	Create Session or Stop TA Service
<i>TEEC_PARAM_TYPES</i>	Set parameter directionality of TEEC_Operation

Table 4.1: CA Operation function of Client API

Client API Data Types

To support structured communication between the CA and TA, OP-TEE defines specific data types for API calls. These data types ensure that data is exchanged securely and predictably. Table 4.2 provides a detailed overview of all data types and their scope within OP-TEE.

Struct	Description
<i>TEEC_Result</i>	Defined to contain the return code that is the result of calling a TEE Client API function
<i>TEEC_UUID</i>	Contains a UUID type defined in RFC4122 and used to identify the TA
<i>TEEC_Context</i>	Logical container that associates the CA with a specific TEE
<i>TEEC_Session</i>	Logical container linking a CA with a particular TA
<i>TEEC_SharedMemory</i>	CA memory blocks registered or allocated to shared memory
<i>TEEC_TempMemoryReference</i>	Shared memory temporarily created by TA service requests
<i>TEEC_RegisteredMemoryReference</i>	Define a memory reference to use some of TEEC_SharedMemory for TA service requests
<i>TEEC_Value</i>	32-bit unsigned integer that does not refer to shared memory but it is passed by value
<i>TEEC_Parameter</i>	Define the parameters of TEEC_Operation
<i>TEEC_Operation</i>	Define TEEC_Parameter and Data Delivery Direction

Table 4.2: Data Types of Client API

Pseudo TAs

Apart from standard Trusted Application, OP-TEE also supports Pseudo TAs, which are integrated directly into the OP-TEE os kernel. Unlike regular TAs, these do not rely on GlobalPlatform APIs but instead use OP-TEE's internal APIs. While pseudo-TAs come with certain restrictions, they are useful for executing privileged operations within the TEE efficiently.

4.1.3 Security and Protection of Trusted Applications

Ensuring the integrity of TAs is a fundamental aspect of OP-TEE's security model. To prevent unauthorized code execution, OP-TEE signs each TA using a private RSA key. However, a current limitation of the toolchain is that all TAs are signed with the same device key, rather than individual keys per TA. Despite this, OP-TEE enforces strict signature verification during the TA loading process, ensuring that only properly signed applications are executed within the TEE.

4.1.4 OP-TEE APIs and Advanced Features

Trusted Application Structure and Implementation

To implement a TA, OP-TEE follows a standardized structure that defines how the TA should be built and which functions it must implement.

Each TA must provide interface functions that act as entry points for handling client requests. These functions include session initialization, command handling, and session termination. OP-TEE provides a framework that standardizes this process, ensuring that TAs interact consistently with the Secure World. Table 4.3 provides an overview of the essential interface functions required for the proper implementation of a TA.

Function	Description
<i>TA_CreateEntryPoint</i>	Run the first time the CA to TA connection is run
<i>TA_OpenSessionEntryPoint</i>	Run the first time the CA to TA connection is run
<i>TA_InvokeCommandEntryPoint</i>	Paired with TEEC_InvokeCommand to provide the service in according to the function or service ID of the TA
<i>TA_CloseSessionEntryPoint</i>	Paired with TEEC_CloseSession to disconnect CA and TA
<i>TA_DestroyEntryPoint</i>	Run when CA to TA is completely terminated

Table 4.3: TA Interface function of internal API

Secure Storage API and Data Encryption

One of OP-TEE's key security features is its secure storage system, designed to encrypt and protect sensitive data using a three-tier system:

- The *Secure Storage Key* (SSK) is derived from the device's unique hardware key
- The *Trusted Application Storage Key* (TSK) is generated for each TA using the SSK and the TA's UUID

- The *File Encryption Key* (FEK) is dynamically created using a pseudorandom number generator (PRNG) for every stored file

By layering these encryption mechanisms, OP-TEE ensures that sensitive data remains secure, even if the REE storage is compromised. Encrypted data is transferred to the tee-supplciant via Remote Procedure Calls (RPCs) and securely stored in protected files.

Additional Libraries for Secure Operations

Beyond secure storage, OP-TEE includes various specialized libraries to support TLS and SSL encryption (*libmbedtls*), mathematical computations (*libmpa*), and a subset of standard C functions (*libututils*). These libraries allow developers to implement secure communication, cryptographic functions, and optimized arithmetic operations within TAs.

4.2 Tpm2-tss

To enable standardized interaction with TPM across different hardware and operating systems, the Trusted Computing Group developed the TPM Software Stack (TSS). This software stack provides various APIs that allow applications to access TPM functionalities in a structured way.

The tpm2-tss is an open-source library that implements the TPM2.0 stack in compliance with TCG specifications. The project provides a set of APIs for interacting with TPM at different levels of complexity. The System API (SAPI) allows direct access to TPM commands, providing maximum control but requiring detailed operation management. The Enhanced System API (ESAPI) simplifies SAPI usage by offering a more intuitive interface for applications. Finally, the Feature API (FAPI) represents the highest level of abstraction, providing an easy-to-use interface for common cryptographic operations.

4.2.1 Tpm2-tss Architecture

The tpm2-tss stack is structured into different layers, each responsible for a specific aspect of TPM communication. The *Transmission Interface* (TCTI) serves as the bridge between software and the TPM hardware or its software emulator, abstracting transport mechanisms and ensuring flexible access. The *System API* (SAPI) provides a direct interface with TPM commands, establishing a one-to-one correspondence between function calls and TPM operations, granting extensive control but requiring detailed knowledge of TPM internals.

Building on this, the *Enhanced System API* (ESAPI) simplifies interactions by handling session data, policies, and cryptographic functions. It is designed to reduce complexity of TPM operations while still allowing significant control over security processes. At highest level, the *Feature API* (FAPI) offers a streamlined abstraction that enables developers to execute cryptographic operations and manage keys without delving into lower-level TPM functionalities.

Each layer works cohesively, ensuring that applications and operating systems can integrate TPM-based security mechanisms efficiently. Whether detailed control over TPM operations is required or high-level cryptographic functionalities are preferred, the tpm2-tss stack provides a structured and accessible approach to leveraging TPM security features.

4.2.2 Role of Tpm2-tss

The TPM was originally conceived as a passive cryptographic processor designed to protect the system from software-based threats. Thanks to the TSS, TPM can be used not only to ensure system integrity but also to provide advanced security functionalities such as authentication and encryption.

The official project documentation is available in the tpm2-tss Github repository, where detailed information about APIs, practical examples, and implementation guidelines can be found. Additionally, the tpm2-software community provides a series of tutorials and resources to facilitate the integration of this stack into existing applications.

One of the most critical elements within tpm2-tss is the Enhanced System API (ESAPI) layer, which plays an essential role in managing cryptography, session data, and policies. This layer is divided into two main parts. The first part is the API component, which defines functions that directly correspond to TPM commands. For example, the *Esys_Create* function in TSS directly maps to the *TPM2_Create* command of the TPM. The second part is the back-end, which implements the core functionalities of ESAPI. Each API invokes several back-end functions to process command parameters before interacting with lower layers and ultimately with the TPM itself.

The ESAPI layer relies on a structure called *ESYS_CONTEXT*, which stores data between calls without maintaining a global state. This structure, defined for external applications as an opaque entity, includes all necessary information for TPM communication, metadata for TPM resources, and operational state data. The specification does not impose a strict data structure, allowing developers to define an appropriate implementation for their needs.

4.2.3 Benefits of Tpm2-tss

The modularity of the tpm2-tss stack enables clear and separate management of TPM functionalities. Its flexibility allows developers to access TPM through different API levels based on their requirements. Due to the standardization of APIs, interaction with TPM remains consistent regardless of the underlying hardware or operating system. Additionally, advanced security ensures the protection of cryptographic keys and secure management of sessions.

4.3 Tpm2-tools

Tpm2-tools is a suite of command-line tools designed to interact with TPM2.0 devices. This library enables users to leverage the security functionalities of the TPM, such as encryption, key generation, identity management, and system integrity protection. With these tools, advanced operations can be executed without the need to write specific code, significantly simplifying TPM management.

4.3.1 Features of Tpm2-tools

Tpm2-tools are implemented using the Trusted Software Stack (TSS) provided by the tpm2-tss project, which serves as the foundation for TPM communication and control. Tpm2-tools allows access to almost all the available TPM functionalities via shell commands and scripts, streamlining automation and integration into various environments. Therefore, tpm2-tools acts as a Command-Line Interface (CLI) wrapper for tpm2-tss, providing a simplified interface for interacting with the TPM module through command-line commands. In particular, in this part, the key features of tpm2-tools are. Table 4.4 summarizes the main commands that provide those security features.

Cryptographic Key Management

Tpm2-tools allows users to create and manage RSA and ECC keys within the TPM. Users can import existing keys, generate new ones, and store them securely within the module. This ensures the protection of cryptographic keys, restricting their use to authorized devices only.

Digital Signature and Verification

Another crucial function provided by the suite is the ability to sign data digitally using keys protected by the TPM. The module enables the generation of reliable signatures and their verification, ensuring the integrity and authenticity of signed data.

Data Protection: Sealing and Encryption

Tpm2 tools include tools for protecting sensitive information through data sealing and encryption. Sealing allows data to be accessible only under a specific TPM software state. This mechanism helps prevent unauthorized access to critical information.

Attestation and Quote Generation

The TPM can be used to collect cryptographic measurements of the system's state and generate quotes that attest to the integrity of the device. This function is particularly useful for ensuring that a system has not been compromised before being authorized to access a secure network.

Access Policy Definition

Tpm2-tools provides the ability to create and manage access policies to control the use of keys and cryptographic operations. These policies can be based on password, or other conditions defined by the user, enabling advanced security management.

TPM Initialization and Management

The library also includes tools for initializing, configuring, and resetting the TPM. These tools allow users to configure the module to fit the system's requirements, ensuring optimal use of its security functionalities.

4.4 Liboqs

The *liboqs* library is a set of post-quantum algorithms developed as part of the Open Quantum Safe project [76]. It is an open-source and free C library that implements cryptographic algorithms designed to withstand quantum attacks. The library includes key encapsulation mechanisms (KEMs) and post-quantum digital signature algorithms. Additionally, it provides a standardized API to facilitate the integration of these algorithms into existing software, along with a testing and benchmarking system to assess their performance. The library was designed as a testing and evaluation platform for researchers and developers, allowing them to experiment with a range of cryptographic schemes believed to be resistant to attacks from quantum computers.

Command	Description
<i>tpm2_startup</i>	Initializes the TPM
<i>tpm2_clear</i>	Resets the TPM to factory settings
<i>tpm2_createprimary</i>	Creates a primary key within the TPM
<i>tpm2_create</i>	Creates a new key within the TPM
<i>tpm2_load</i>	Loads a key into the TPM
<i>tpm2_evictcontrol</i>	Makes a key persistent within the TPM
<i>tpm2_sign</i>	Signs a file using a TPM key
<i>tpm2_verifysignature</i>	Verifies a digital signature
<i>tpm2_seal</i>	Seals data within the TPM
<i>tpm2_unseal</i>	Unseals previously sealed data
<i>tpm2_quote</i>	Generates a quote
<i>tpm2_pcrread</i>	Reads the value of a PCR
<i>tpm2_encryptdecrypt</i>	Encrypts and decrypts data using the TPM
<i>tpm2_policypassword</i>	Creates a policy based on a password

Table 4.4: Main Commands of tpm2-tools

4.4.1 Supported Algorithms

The liboqs library implements numerous algorithms that have been proposed for standardization in the NIST Post-Quantum Cryptography Standardization Process. These algorithms are classified based on the security levels defined by NIST, ranging from Level 1, which provides minimal protection, to Level 5, which ensures the highest level of security. The *NIST security level* is a criterion used to evaluate the robustness of cryptographic algorithms against both classical and quantum attacks. This classification helps determine which algorithm to adopt based on the required protection level.

Table 4.5 provides a detailed overview of the available algorithms [77], indicating security levels and their specific characteristics, including the sizes of public keys, secret keys, and signatures.

Algorithm	Claimed NIST Level	Public Key Size (Bytes)	Secret Key Size (Bytes)	Signature Size (Bytes)
Dilithium2	2	1312	2528	2420
Dilithium3	3	1952	4000	3293
Dilithium5	5	2592	4864	4595
Falcon-512	1	897	1281	752
Falcon-1024	5	1793	2305	1462
Falcon-padded-512	1	897	1281	666
Falcon-padded-1024	5	1793	2305	1280
ML-DSA-44	2	1312	2560	2420
ML-DSA-65	3	1952	4032	3309
ML-DSA-87	5	2592	4896	4627
SPHINCS+-SHA2-128f-simple	1	32	64	17088
SPHINCS+-SHA2-128s-simple	1	32	64	7856
SPHINCS+-SHA2-192f-simple	3	48	96	35664
SPHINCS+-SHA2-192s-simple	3	48	96	16224
SPHINCS+-SHA2-256f-simple	5	64	128	49856
SPHINCS+-SHA2-256s-simple	5	64	128	29792
SPHINCS+-SHAKE-128f-simple	1	32	64	17088
SPHINCS+-SHAKE-128s-simple	1	32	64	7856
SPHINCS+-SHAKE-192f-simple	3	48	96	35664
SPHINCS+-SHAKE-192s-simple	3	48	96	16224
SPHINCS+-SHAKE-256f-simple	5	64	128	49856
SPHINCS+-SHAKE-256s-simple	5	64	128	29792

Table 4.5: Available PQ Algorithms for digital signatures and Specifications

Chapter 5

Design and Implementation

This chapter aims to provide an overview of the design choices made to integrate fTPM into OP-TEE and introduce support for post-quantum algorithms. The architectural and technological decisions behind the project will be analyzed, highlighting the advantages and limitations of the adopted solutions.

5.1 Design Choices

During the implementation process of the fTPM in OP-TEE, several alternatives were considered to ensure an optimal balance between security, efficiency, and compatibility. One of the most important aspects was identifying the most suitable execution environment. Among the available options, OP-TEE proved to be the most advantageous choice due to its open-source nature and its compatibility with ARM TrustZone, allowing seamless integration with a secure execution environment.

The use of a hardware TPM was ruled out since there was no access to a physical TPM. This led to the selection of fTPM as an alternative solution, allowing the implementation of a software architecture capable of emulating TPM functionalities without requiring dedicated hardware. The approach not only ensured compatibility with modern cryptographic systems but also provided greater flexibility in managing TPM functionalities within the OP-TEE environment. For the fTPM source code, we opted for Microsoft's implementation, as it is open source and compatible with OP-TEE, ensuring smoother integration and a well-documented and supported codebase.

The selection of the most suitable post-quantum also required in-depth analysis. Algorithms such as Dilithium and Kyber were considered. Still, ultimately, SPHINCS-SHAKE-256f-simple was chosen for its robustness based on hash functions, making it independent of mathematical problems that could be vulnerable to quantum attacks. The choice was further motivated by the fact that, despite its large key size and slow signature generation, it provides the highest security guarantees among post-quantum algorithms. For a detailed discussion of the criteria and reasoning behind the algorithm selection, refer to Section 2.7, where the post-quantum algorithm evaluation is covered comprehensively.

Additionally, the integration of the liboqs library, which supports a wide range of post-quantum algorithms, was evaluated. One of the key advantages of liboqs is its modular nature, which allows easy integration and testing of different algorithms without requiring individual implementations. Liboqs remains an interesting solution for those looking to experiment with various post-quantum algorithms without having to develop them from scratch.

5.1.1 General Architecture

The fTPM implementation was developed as an early-TA (Early Trusted Application), a mode that allows the application to be loaded and initialized during the system boot phase. This approach ensures that TPM functionalities are immediately available, avoiding any latency in accessing critical security services.

In OP-TEE, early TAs are not a new concept but a key mechanism for ensuring security operations right from the system boot. For example, there are early TAs dedicated to boot authentication and key initialization, essential for establishing a secure environment from the very start of the device. The integrity of early TAs is ensured through the authentication of the entire firmware, which prevents unauthorized code execution and protects the system from potential compromises.

All early TAs, including the fTPM implementation, are executed entirely within the Secure World of OP-TEE. This design choice is fundamental for ensuring a high level of security, as it prevents execution within the Normal World, where applications are exposed to potential threats and exploits. Running early TAs in the Secure World guarantees that sensitive operations, such as cryptographic key handling and authentication mechanisms, remain isolated from potentially compromised components of the system. This isolation mitigates attack vectors, including privilege escalation, code injection, and memory tampering, making the entire boot and initialization process more resilient against security threats.

The presence of fTPM from the boot phase is essential because it ensures that security-critical services are available from the earliest stages of system execution. This prevents scenarios where an untrusted or compromised environment could interfere with TPM operations before it is fully initialized. By integrating fTPM as an early TA, secure storage, key management, and authentication mechanisms are established before the Normal World is even operational, eliminating a potential attack window. This also allows the enforcement of secure boot mechanisms, ensuring that only signed and verified firmware and software components are executed, further reducing the risk of tampering and unauthorized modifications.

Figure 5.1 illustrates the project’s framework, detailing the interaction between the secure and non-secure components and their integration within a remote attestation context.

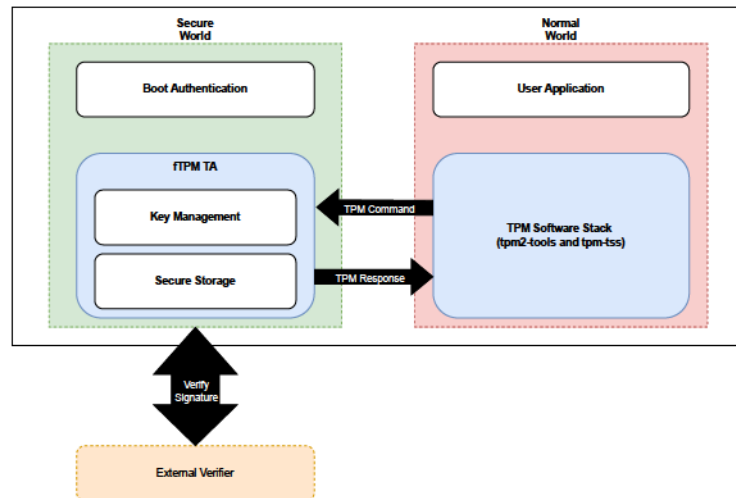


Figure 5.1: Implementation Architecture

5.2 Technological Choices

The choice of OP-TEE as the Trusted Execution Environment was motivated by its open-source nature and its compatibility with ARM TrustZone, ensuring security at the hardware level. Furthermore, OP-TEE adheres to GlobalPlatform standards, making it a robust and widely adopted solution for trusted applications. For classical cryptographic management, mbedTLS was used, while fTPM was modified to be executed as an early TA, leveraging OP-TEE's capabilities to the fullest.

One of the most relevant aspects of the project was adapting the tpm2-tools and tpm2-tss libraries to the new algorithm and allowing TPM command management directly from the Rich Execution Environment (REE). This modification improved the interoperability between the virtual TPM and the management tools available in the system, facilitating TPM use and configuration across different environments.

The tpm2-tools and tpm2-tss libraries play a crucial role in interacting with a TPM 2.0. Tpm2-tools provides a set of command-line tools that enable cryptographic operations, key management, and security enhancements. Tpm2-tss, on other hand, implements the Trusted Software Stack for TPM 2.0, facilitating TPM integration with other applications and allowing more detailed control over low-level operations.

5.3 Project Structure

. The overall structure of the project is the following:

```
optee_ftpm
├── build
├── buildroot
├── liboqs
├── linux
├── ms-tpm-20-ref
├── optee_client
├── optee_examples
├── optee_ftpm
├── optee_os
├── optee_test
├── out
├── out-br
├── qemu
├── toolchains
├── tpm2_tools
├── tpm2_tss
├── trusted-firmware-a
└── u-boot
```

5.4 Ms-tpm-20-ref

Microsoft has made available `ms-tpm-20-ref`, an open-source reference implementation of TPM2.0 primarily intended for educational and development purposes. It was specifically chosen for this implementation due to its compatibility with OP-TEE, enabling seamless integration with the trusted execution environment.

5.4.1 Project Architecture and Structure

The `ms-tpm-20-ref` project is structured around three main components. The TPM core manages internal states and TPM hierarchies, ensuring the correct execution of security operations. The cryptographic modules support various algorithms, including RSA, ECC, AES, and SHA-256 hash functions, providing the necessary support for data protection. Finally, the simulator allows testing the TPM's behavior without the need for dedicated hardware, facilitating software-based development and experimentation.

The source code is organized hierarchically to ensure clear and modular management of functionalities. In this implementation, particular attention has been given to the *TPMCmd* directory, which contains the source code for fTPM and serves as the core of cryptographic operations and key management. The integration of new post-quantum algorithms required the introduction of four new data structures specifically designed for managing public and private keys, along with two structures dedicated to digital signature management. To ensure the proper integration of these structures, Marshalling, and Unmarshalling functions were developed to serialize and deserialize information in TPM data flows.

5.4.2 Execution in OP-TEE

In this implementation, the simulator provided by the `ms-tpm-20-ref` project was not used, as the code was adapted to run directly in OP-TEE as an Early Trusted Application (early TA). This approach allowed for realistic testing in a protected environment, ensuring a higher fidelity to actual operational conditions.

5.4.3 Cryptographic Function Extension

To make new post-quantum algorithms directly available within the TPM, the `liboqs` library was integrated, significantly expanding the cryptographic capabilities available in the TPM. During this implementation, high-level TPM functions such as *TPM2_CreatePrimary*, *TPM2_Create*, *TPM2_Sign*, and *TPM2_Quote* were not directly modified. However, necessary cryptographic management functions such as *CryptCreateObject* for object creation and *CryptSign* for digital signature generation were extended to include support for the newly introduced algorithm.

5.4.4 TPM Hierarchies

TPM2.0 defines three fundamental hierarchies: Owner, Endorsement, and Platform. The Owner hierarchy is responsible for the administrative management of the TPM, allowing for policy definition and key management. The Endorsement hierarchy ensures the unique identification and attestation of the TPM, providing a trust base for cryptographic operations. Finally, the Platform hierarchy manages the control of the hardware platform and ensures the integrity of the boot process, contributing to a secure environment from system setup. The three hierarchies are summarized in the Table 5.1.

Internally, these hierarchies are represented by cryptographic seeds, which are initialized during the boot phase or retrieved from the non-volatile memory of the hardware board, if available. In this implementation, it was necessary to fix the Endorsement hierarchy seed, known as the Endorsement Primary Seed (EPSeed), to ensure deterministic generation of an Endorsement Key

Hierarchy	Function
Owner	TPM management, policy definition and key management
Endorsement	Unique TPM identification and attestation
Platform	Hardware platform control and boot integrity

Table 5.1: TPM Hierarchies in TPM2.0 specification

(EK). This key must remain constant to allow the creation of a stable and reliable certificate. However, the seed value varies from board to board, depending on the specific value injected by the manufacturer into the hardware.

5.5 Optee_fTPM

This is the directory in which fTPM TA is contained. In this sense, the fTPM implementation was developed as an Early Trusted Application (early TA), a mode that allows the application to be loaded and initialized during the system boot phase. This approach ensures that TPM functionalities are immediately available, avoiding latency in accessing critical services.

In OP-TEE, early TAs are a key mechanism for ensuring security operations right from the system boot. For example, there are early TAs dedicated to boot authentication and key initialization, essential for establishing a secure environment from the very start of the device. The integrity of early TAs is ensured through firmware, which prevents unauthorized code execution and protects the system from potential compromises.

All early TAs, including the fTPM implementation, are executed entirely within the Secure World of OP-TEE. This design choice is fundamental for ensuring a high level of security, as it prevents execution within the Normal World, where applications are exposed to potential threats and exploits. Running early TAs early TA in the Secure World guarantees that sensitive operations, such as cryptographic key handling and authentication mechanisms, remain isolated from potentially compromised components of the system. This isolation mitigates attack vectors, including privilege escalation, code injection, and memory tampering, making the entire boot and initialization process more resilient against security threats.

The adoption of an early TA for fTPM has proven advantageous in several ways. It ensures that the TPM is operational from the very beginning, eliminating the need for late initialization, which could expose the system to vulnerabilities.

Additionally, it provides stronger protection for critical data, as the application is launched when the system is still in a controlled and secure state. Finally, direct integration with the firmware allows a smoother interaction between the TPM and another system component.

5.6 Tpm2-tss and Tpm2-tools

The integration of post-quantum algorithms into an fTPM required modifications to the tpm2-tss and tpm2-tools libraries, which are fundamental for interfacing with the TPM. These libraries respectively provide support for the TPM2.0 software layer and a set of tools for interacting with the hardware or software module.

5.6.1 Reasons for modifications

These modifications were primarily necessary to ensure support for post-quantum algorithms, as the original libraries did not include definitions and functions compatible with these new cryptographic standards. Furthermore, some internal APIs of tpm2-tss did not account for the

management of keys and signatures using post-quantum algorithms, necessitating a revision of the command structure and hashing functions. Additionally, the tools in the `tpm2-tools` were modified to allow the generation and importation of post-quantum keys, expanding their capabilities. Finally, new test routines were introduced to verify the proper functioning of cryptographic operations, ensuring reliability and stability within the TPM.

To enable communication between the REE and the TPM, as well as to verify the correct functionality of post-quantum algorithms, these two libraries were integrated into the implementation. These libraries provide a reliable interface between the application layer and the TPM, facilitating the interaction with keys and commands required for managing post-quantum cryptography.

5.6.2 Technical Details

The modifications affected various areas of the `tpm2-tss` and `tpm2-tools` code.

Tpm2-tss

Specifically, regarding `tpm2-tss`, it was necessary to extend the source files related to key management and digital signatures to include the post-quantum algorithm SPHINCS+. To support this algorithm's specifications, the TPM key formats were updated, and the hashing and signing APIs were adapted to accept and process the new formats. Additionally, new test cases were added to verify the correct functionality of hashing and signing operations using post-quantum algorithms.

Another significant modification involved adding new data structures within `tpm2-tss`, necessary to support these new algorithms. Four new structures were introduced: one for the public key, one for the private key, and two for digital signature. These structures efficiently handle the specific parameters of SPHINCS+ within the TPM architecture, ensuring proper storage and manipulation of the generated keys and signatures. Additionally, these four structures were integrated into existing structures, allowing SPHINCS keys to be included alongside the existing algorithms.

Moreover, a new constant was added to identify the new algorithm. This constant was appended to the long list of supported algorithms and serves to check whether the algorithm is recognized and usable by the system.

Another fundamental aspect concerns the Marshalling and Unmarshalling functions in `tpm2-tss`. In this case, some existing functions were modified, and new ones were added to ensure compatibility and proper data transfer. These functions are essential for the structured conversion of information between the various TPM components, ensuring that post-quantum keys and signatures are correctly serialized and deserialized during key generation and signing operations.

The modifications made to `tpm2-tss` are symmetric to those implemented in the TPM, as ensuring this symmetry is crucial. Without this alignment, the TPM would not be able to recognize and correctly utilize the implemented algorithms, compromising interoperability and proper system functionality.

Tpm2-tools

In the case of `tpm2-tools`, several commands were modified to allow the management of post-quantum keys and signatures. Specifically, the `tpm2_createprimary`, `tpm2_create`, `tpm2_sign`, `tpm2_quote`, `tpm2_createek` and `tpm2_createak` commands were updated to support these new algorithms, introducing specific options. The modifications in `tpm2-tools` also included the propagation of the newly introduced constant, as it affects a switch case responsible for setting algorithm parameters. While in this implementation the addition was only necessary for a single post-quantum algorithm, this change lays the groundwork for future implementations where multiple algorithms with different parameters could be supported.

5.7 Considerations

The integration of fTPM in OP-TEE as an early TA, combined with the adoption of SPHINCS-SHAKE-256f as a post-quantum algorithm, represents a crucial step forward in ensuring the security and resilience of embedded systems against both classical and future quantum threats. By leveraging OP-TEE, a trusted execution environment fully compatible with ARM TrustZone, this implementation provides strong isolation between critical security functions and the rest of the system, minimizing the risk of unauthorized access or tampering.

The choice of Microsoft’s fTPM as the foundation for the implementation ensured an open-source, well-documented, and reliable base, while the adaption of tpm2-tools and tpm2-tss allowed full compatibility with existing TPM-based security mechanisms. The evaluation of liboqs further highlighted the potential for future expansion, demonstrating the flexibility of this approach in integrating advanced cryptographic algorithms.

A key design decision was to execute all early TAs in the Secure World, guaranteeing a strong security posture at system boot. This ensures that all cryptographic operations, key management, and authentication of the entire firmware further reinforce the integrity of the system, preventing unauthorized modifications and mitigating attacks that could compromise the boot process.

While SPHINCS-SHAKE-256f was chosen for its unparalleled security guarantees, its computational cost, and large key sizes remain challenges for resource-constrained environments. However, the trade-off between security and performance was carefully balanced, ensuring the feasibility of this implementation in real-world applications.

Chapter 6

Tests and Results

This chapter will introduce the test that has been performed, detailing their setup, the measured metrics, and their impact on the integration within the fTPM.

6.1 Introduction

The primary objective of these tests is to assess the feasibility, performance, and security of PQ-based cryptographic operations in a constrained environment. The evaluation is conducted through a series of functional and performance tests, ensuring compliance with expected cryptographic standards and TPM specifications.

The testing process involves verifying the correct implementation of PQ algorithms, measuring execution times, analyzing resource consumption, and assessing potential security vulnerabilities. Specifically, the tests ensure the correct integration of the liboqs library with OP-TEE, the proper incorporation of fTPM into the Secure World, and the successful execution of PQ algorithms within the fTPM. Additionally, comparisons with classical cryptographic implementations are performed to evaluate the impact of PQ integration on fTPM efficiency.

For these tests, the *SPHINCS-SHAKE-256f-simple* algorithm was selected due to its lower cryptographic requirements, making it more suitable for integration into modern TPMs, as well as its strong security guarantees and ease of upgradability. The evaluation focuses on PQ key generation and signature generation, comparing these operations in terms of resource consumption and execution time against RSA-2048, one of the most secure cryptographic standards in use today.

The results obtained from these tests will provide valuable insights into the readiness of PQ cryptographic solutions for practical deployment within TPM-based security infrastructures. Furthermore, they will help identify potential bottlenecks and optimization strategies for future implementations.

6.2 Configuration of Test Environment

Configuring an appropriate test environment was essential to ensure reliable and reproducible results. Since no physical development board was available, QEMU was used to create a virtualized environment that emulates an ARMv7 board. This setup allowed for the execution of OP-TEE and an fTPM within the Secure World, simulating conditions as close as possible to those of real hardware. The entire system was tested on Ubuntu 22.04, ensuring compatibility with all required components.

The software stack consisted of various elements necessary for the execution of tests. OP-TEE

served as TEE running in the ARM TrustZone, while the Linux Kernel supported OP-TEE and TPM functionalities. The fTPM implementation was completed by tpm2-tools, a suite of command-line utilities facilitating interaction with the TPM, and tpm2-tss, which enabled the seamless execution of TPM commands while ensuring compatibility. To integrate post-quantum cryptographic algorithms, liboqs were included, providing the required cryptographic functionalities.

During the testing phase, different Trusted Applications (TAs) were deployed within the environment. The liboqs TA was developed to validate the integration of post-quantum algorithms. A separate Benchmarking TA was implemented to measure system performance overhead.

Setting up the environment involved multiple steps. Initially, OP-TEE was compiled and deployed within the QEMU virtual machine. This was followed by configuring the fTPM in the Secure World and installing tpm2-tools and tpm2-tss on the REE to facilitate TPM command execution. The next step involved building and integrating the liboqs library within OP-TEE secure applications to enable post-quantum cryptographic operations. Finally, custom scripts were deployed to automate both functional and performance tests, ensuring efficient execution and data collection.

For accurate measurements, the POSIX library was employed, providing highly precise time counters capable of measuring execution times down to the nanosecond level.

By leveraging this well-configured virtualized test environment, it was possible to conduct extensive and reliable testing without the need for physical hardware. This setup provided a robust platform for evaluating the integration of post-quantum cryptographic algorithms in OP-TEE and fTPM while allowing for in-depth performance analysis under controlled and repeatable conditions.

6.3 Test Methodology

This section describes the testing methodologies adopted to ensure the quality and reliability of integrating post-quantum algorithms within OP-TEE and an fTPM running in the Secure World. The primary goal of the tests is to verify that post-quantum algorithms are correctly implemented, providing an adequate level of security without compromising system performance.

Functional tests were conducted to ensure that post-quantum cryptographic operations were executed correctly, meeting the requirements of integrity, authentication, and sensitive data protection. Specifically, these tests analyzed whether these algorithms could replace or complement traditional cryptographic solutions already present within the OP-TEE and fTPM architecture, ensuring a stable and reliable integration.

In parallel, performance tests evaluated the impact of adopting these new algorithms on the overall system. Since post-quantum security introduces computational overhead, it was crucial to measure the time required for cryptographic operations such as key generation and digital signatures, comparing performance with traditional algorithms like RSA-2048. Beyond this direct comparison, the computational cost of the world switching between Normal World and Secure World was analyzed, along with the necessary metrics to ensure efficient communication between a Client Application (CA) and a Trusted Application (TA).

Through this structured testing approach, it was possible to identify potential integration issues, assess performance sustainability, and optimize the system to guarantee a high level of security with minimal impact on the device's available resources.

6.3.1 Functional Tests

Functional tests verify that the integration of post-quantum algorithms within OP-TEE and fTPM is correctly executed and that the system meets the expected functional requirements.

Two main tests were conducted. The first verified that post-quantum algorithms were properly integrated and operational within the OP-TEE environment. For this test, a Trusted Application (TA) was created following the guidelines in the official OP-TEE documentation [78]. Within the TA, the basic functionalities provided by the liboqs library were tested, including context generation, key generation, signature generation, and signature verification. The objective was to confirm the correct integration of the liboqs library and, consequently, the post-quantum algorithms within OP-TEE. The TA can be invoked from the Rich Execution Environment (REE) using the command `optee_example_liboqs`.

The second test analyzed the addition of post-quantum algorithms to fTPM to ensure they could be utilized in the Secure World. For this phase, shell scripts were developed to test the correct execution of tpm2-tools commands thrown from REE, which were then executed within the fTPM in the Secure World. These scripts simulated various operational scenarios, including generating an endorsement key, an attestation key, a digital signature, and a quote. The goal was to verify the ability to invoke post-quantum algorithms directly from the REE, ensuring that the system could leverage the new cryptographic functionalities even outside the Secure World.

6.3.2 Performance Tests

Performance tests measured the impact of integrating post-quantum algorithms on the performance of OP-TEE and fTPM, analyzing resource consumption and execution times.

In the first performance test, scripts were developed to test the execution of tpm2-tools commands and measure the execution time of each operation. To ensure more accurate results, each command was executed 100 times for both algorithms, collecting all execution times to calculate the mean, standard deviation, and relative error. The tested commands included `tpm2_createprimary`, for generating a primary object such as endorsement key, `tpm2_create`, for creating a child object such as an attestation key, `tpm2_sign`, for digitally signing an object, and `tpm2_quote`, for generating a quote. The objective of this test was to analyze the performance differences between the operations performed with the `sphincs-shake-256f-simple` algorithm and the same operations performed with RSA-2048. This evaluation provides insights into how these algorithms perform under different conditions, highlighting differences in computational requirements and execution time.

In the second performance test, another TA was created, callable from a CA via the `optee_example_benchmark` command. This TA is empty and serves solely to measure the overhead introduced when launching a TA from the REE. The necessary steps for the launch of the TA and the establishment of the connection between the CA and the TA were analyzed and shown in Figure 6.1. On the CA side, the time required to create the context, open the session, switch worlds, and close the context and session was measured. To obtain more precise measurements, each command was executed 100 times, collecting data to calculate the average time, standard deviation, and relative error.

6.4 Results

This section presents the results of the performance tests. As stated before, the testing process was designed to analyze two key aspects: the computational performance of key and signature generation for both cryptographic schemes and the additional overhead introduced by OP-TEE when interacting with the Client Application (CA). This evaluation is particularly relevant for understanding the cost of the world switch operation, which is a critical factor in ARM-based

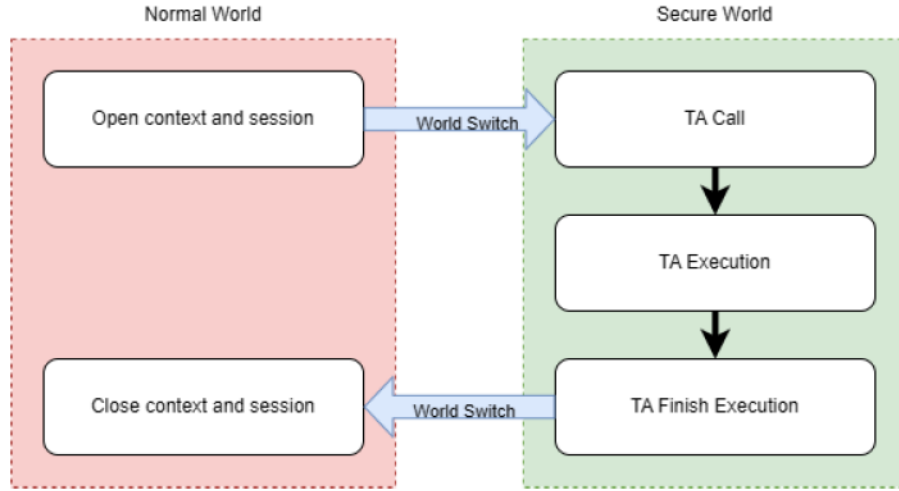


Figure 6.1: Example of execution of a CA

systems.

The results obtained from these tests contribute to assessing the feasibility of using SPHINCS and RSA in security-critical applications, considering both performance and operational constraints. The following sections detail the findings of each test category and provide a comparative analysis of their impact on system performance.

6.4.1 Key and Signature Generation Performance Analysis

To better analyze the differences in performance between SPHINCS and RSA, the test was conducted by simulating a digital signature operation, measuring the computational cost of each step. This approach allowed for an in-depth analysis of the most time-consuming operations in both algorithms, highlighting their respective bottlenecks. Figure 6.2 further illustrates these differ-

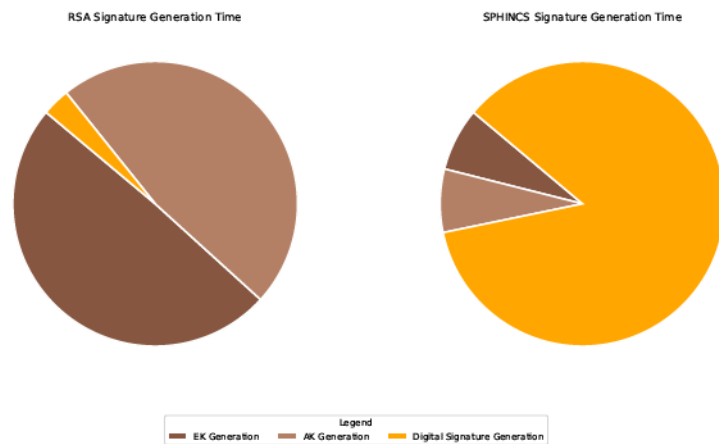


Figure 6.2: Comparison between RSA and SPHINCS signing operations

ences. Even though the total execution times of the two algorithms are vastly different, analyzing the individual operations reveals critical insights. In RSA, the most computationally expensive operation is key generation, which accounts for 96.9% of the total process. This is due to the complexity of generating large prime numbers and verifying their cryptographic strength, which requires extensive probabilistic tests. In contrast, in SPHINCS, the signing process dominates the

execution time, contributing to 85.6% of the entire procedure. This is primarily due to the intensive hash function computations required to produce a SPHINCS signature, which significantly increases the computational cost.

Examining the numerical results, SPHINCS demonstrated a significant advantage in the generation, completing the creation of an Endorsement Key (EK) in 8.30 seconds and an Attestation Key (AK) in 8.31 seconds, both with a 1% relative error. On the other hand, RSA exhibited considerably higher key generation times, with the EK taking 33.84 seconds and the AK requiring 32.55 seconds.

The signature generation results reveal a stark contrast. SPHINCS required 97.10 seconds to generate a signature, whereas RSA completed the same task in just 2.15 seconds, achieving a 98% efficiency gain. The reason for this drastic difference lies in the fundamental design of the two algorithms. SPHINCS, being a post-quantum cryptography scheme, relies on a hash-based approach, necessitating multiple layers of hash function evaluations to construct a secure signature. This significantly inflates computational requirements, making SPHINCS impractical for real-time applications. Conversely, RSA employs modular exponentiation, which, despite its computational cost, is highly optimized in modern processors, allowing it to generate signatures at a significantly faster rate. All numerical values are summarized in Table 6.1. These findings high-

Feature	SPHINCS-SHAKE-256f-simple	RSA-2048
EK Generation	8.30 s	33.84 s
AK Generation	8.31 s	32.55 s
Digital Signature	97.10 s	2.15 s
Total	113.71 s	69.54 s

Table 6.1: Comparison cryptographic operations between SPHINCS-SHAKE-256f-simple and RSA-2048

light the trade-offs between security and efficiency. SPHINCS offers superior resilience against quantum attacks, making it a robust choice for future cryptographic needs, but at the cost of significantly higher signature generation times. RSA, while much faster, remains vulnerable to quantum threats and may eventually be replaced by post-quantum alternatives. The observed performance trends indicate that, for applications prioritizing fast cryptographic operations, RSA remains the preferable choice, while SPHINCS should be reserved for scenarios where quantum security is a primary concern.

6.4.2 OP-TEE Overhead

The second test was conducted independently to analyze the performance impact of invoking a Trusted Application (TA) from a Client Application (CA) within OP-TEE. Unlike the cryptographic performance evaluation, this test specifically aimed to measure the computational overhead introduced by OP-TEE when performing a world switch operation, a fundamental aspect of ARM-based architectures.

The evaluation considered multiple factors contributing to the overall overhead. The creation of the execution context was measured at 542 microseconds, followed by session opening, which took 365 milliseconds, making it the most time-consuming operation in this process. The world switch itself, which represents the transition between the Normal World and the Secure World, had an overhead of 827 microseconds. Once the operation was completed, closing the session required 3.31 milliseconds, while the final step of closing execution context was measured at 82.7 microseconds. All measured data related to OP-TEE invocation overhead have been presented in tabular format in Table 6.2 to facilitate analysis.

From the measured execution time of the world switch, the number of CPU cycles required for the

operation was calculated. To estimate this, the BogoMIPS variable was introduced. BogoMIPS is an approximated system-calculated value that represents the frequency of the processor, providing a rough estimation of the number of million instructions per second that the CPU can execute. By leveraging this value, it was possible to compute the number of cycles required for the world switch operation, offering insight into the actual processing cost.

The calculation resulted in 51679 CPU cycles for the world switch operation. Comparing this result to state-of-the-art expected values, which range between 50000 and 90000 cycles, confirms that OP-TEE’s performance aligns with theoretical expectations. However, while the results fall within the expected range, OP-TEE remains significantly less efficient than its Intel SGX counterpart, where the same operation requires seven times fewer cycles. This efficiency gap underscores the architectural differences between ARM-based and x86-based trusted execution environments and highlights the necessity of further optimizations in OP-TEE’s secure execution transitions.

Feature	Execution Time
Open Context	542 μs
Open Session	365 ms
World Switch	827 μs
Close Session	3.31 ms
Close Context	82.7 μs

Table 6.2: Analysis overhead in OP-TEE

6.5 Comparison and Final Observations

The results of this study confirm that RSA offers significantly better performance in key and signature generation, making it less practical for applications requiring high-speed operations. RSA, on the other hand, benefits from well-established optimizations that allow for faster execution, making it more suitable for real-time or resource-constrained environments.

The second test on OP-TEE overhead revealed that the world switch operation plays a crucial role in secure application performance. While OP-TEE’s switching performance aligns with expected values, its efficiency is still limited when compared to Intel SGX, which can complete the same transition in seven times fewer CPU cycles. The high overhead associated with session initialization and world switching indicates the need for optimizations in OP-TEE’s secure execution framework, particularly in ARM-based architectures where frequent secure/non-secure transitions can introduce significant delays.

From a broader perspective, these findings suggest that the choice between SPHINCS and RSA should be driven by the balance between security and performance. SPHINCS is essential for future-proof cryptographic security, especially in a post-quantum era, but its computational demands must be carefully considered. RSA, while faster and more efficient, remains vulnerable to future quantum computing threats and may eventually be phased out in favor of quantum-resistant solutions.

For real-time applications, SPHINCS is not an optimal solution due to its high signing costs. However, it remains the most secure option against quantum computing threats. A possible alternative for real-time applications is Kyber-Dilithium, which provides a trade-off between security and performance, offering a viable middle ground for systems requiring both speed and future-proof cryptographic strength.

For OP-TEE, reducing world switch latency and session overhead would be beneficial for improving system efficiency, particularly in embedded and real-time systems where performance

constraints are critical. Future work should focus on further optimizing the secure world transition mechanisms, exploring hardware acceleration strategies, and refining session handling to minimize computational delays.

Overall, these results provide valuable insights into the trade-offs between cryptographic performance and secure execution overhead, highlighting areas for future research and development in both cryptographic algorithms and trusted execution environments.

Chapter 7

Conclusions

This thesis explored the integration of post-quantum algorithms within a Trusted Platform Module (TPM), addressing the challenges related to security, efficiency, and compatibility with embedded environments. In particular, the work focused on implementing a firmware TPM (fTPM) in OP-TEE, a Trusted Execution Environment compatible with ARM Trust Zone, and introducing cryptographic algorithms resistant to quantum attacks.

One of the central aspects that emerged concerns the choice of OP-TEE as the execution platform for fTPM. This decision ensured seamless integration with a secure execution environment, leveraging its compatibility with ARM TrustZone and adherence to GlobalPlatform standards. Furthermore, implementing fTPM as an Early Trusted Application (early-TA) strengthened security by ensuring that critical services were available from the earliest system boot stages, reducing potential attack vectors.

Several candidates were evaluated regarding the choice of post-quantum algorithms, including Dilithium and Kyber. However, SPHINCS-SHAKE-256f-simple was selected primarily due to its strong security guarantees, prioritizing security over performance. Its robustness, based on hash functions, eliminates dependencies on mathematical problems that could be vulnerable to quantum attacks. Although it has some limitations in terms of key size and signature generation speed, its high security makes it a reliable option for TPM integration.

The implementation also involved adapting the tpm2-tools and tpm2-tss libraries to ensure compatibility with the new algorithms. These modifications improved interoperability between the virtual TPM and available management tools, facilitating usage and configuration across different environments.

The conducted tests aimed to assess the feasibility, performance, and security of post-quantum cryptographic operations in a resource-constrained environment. Functional and performance tests were carried out to ensure proper integration of the algorithms, measuring execution times and resource consumption. In particular, the integration of the liboqs library with OP-TEE and the inclusion of fTPM in the Secure World were successfully verified.

The results showed that SPHINCS-SHAKE-256f-simple offers advantages in terms of security but has significant performance limitations. Key generation proved to be more efficient than RSA-2048, while signature generation required significantly more time. This difference is not due to the cost of hash functions per se, but rather to the need to reconstruct the entire Merkle Tree during the signing process and the large size of the resulting signature, both of which contribute substantially to the computational overhead.

The comparative analysis with RSA-2048 highlighted a trade-off between security and performance: while SPHINCS protects against quantum attacks, its high latency makes it difficult to use in real-time applications. The analysis of world-switching metrics between the Normal World and the Secure World also enabled the evaluation of the efficiency of communication between

management applications and the virtualized TPM.

Despite the obtained results, the research presents some limitations. The absence of a hardware TPM necessitated the exclusive use of a software implementation, which may not fully reflect the challenges associated with adopting post-quantum algorithms on real devices. Additionally, the performance of SPHINCS-SHAKE-256f-simple, particularly its signature generation times, represents a constraint to consider in real-time applications. For this reason, in scenarios where performance is a critical factor, it may be preferable to evaluate the use of algorithms such as Dilithium, which offers a better balance between security and execution speed.

In the future, further research could focus on optimizing the performance of post-quantum algorithms in TPMs and integrating them with hardware TPMs when available. Evaluating optimization strategies to reduce signature generation times could make post-quantum algorithms more feasible for scenarios with stringent time constraints. Another future step could be integrating additional post-quantum algorithms, expanding the available choices, and allowing the selection of the most suitable solution based on security and performance requirements.

This thesis aims to provide an initial solution to the quantum threat, which is expected to become a reality within less than a decade. The work conducted represents a first step toward the transition to secure systems against quantum attacks, offering a significant contribution to research and future developments in cybersecurity. Additionally, this research demonstrated that it is possible to integrate post-quantum algorithms within modern specifications while maintaining compatibility with current standards, ensuring a gradual transition to the post-quantum era without compromising existing infrastructures.

Overall, this work demonstrates how a well-structured, modular, and secure TPM implementation in OP-TEE can provide a robust foundation for future embedded systems. As cryptographic standards continue to evolve, the flexibility of this approach will allow for future improvements, ensuring long-term security and adaptability. Future work may focus on optimizing performance, integrating additional post-quantum algorithms, and further enhancing hardware acceleration capabilities to meet the demands of next-generation secure computing environments.

In conclusion, this thesis contributed to demonstrating the feasibility of integrating post-quantum algorithms into TPMs, providing a foundation for the transition to systems resistant to quantum attacks. The identified challenges and proposed solutions offer valuable insights for future developments and strengthening security in embedded devices.

Appendix A

User Manual

This user manual provides a practical guide for interacting with a firmware-based Trusted Platform Module (fTPM) within OP-TEE Trusted Execution Environment.

This manual assumes all configuration and build steps described in Appendix B have been completed. The procedures described here refer to the implementation developed as part of this thesis project, which integrates an fTPM Trusted Application into the OP-TEE framework.

A.1 Requirements

- OP-TEE ([GitHub OP-TEE](#))
- Liboqs ([GitHub Liboqs](#))
- Linux OS

A.2 Build Steps

A.2.1 Install Dependencies

Install dependencies on Ubuntu:

```
$ sudo apt install device-tree-compiler bison flex libssl-dev g++  
libgmp-dev libmpc-dev ninja-build pkg-config libglib2.0-dev  
libpixman-1-dev meson cmake gcc python3-pytest python3-pytest-xdist  
unzip xsltproc doxygen graphviz python3-yaml valgrind
```

A.2.2 Build Project

Once all dependencies are installed, compile the project using the following command:

```
cd build  
make run MEASURED_BOOT_FTPM=y
```

This command must be executed from the *build* directory, where all build targets are defined. The `MEASURED_BOOT_FTPM` option enables the integration of the fTPM during the build process and ensures its initialization after boot.

During the first compilation, the process may take a significant amount of time, in some cases up to an hour.

A.3 fTPM Use

After executing the command, QEMU will launch along with two UART consoles: one for the Secure World and one for the Normal World. The main QEMU console will pause, waiting for user input. To proceed with the system boot, enter the following command:

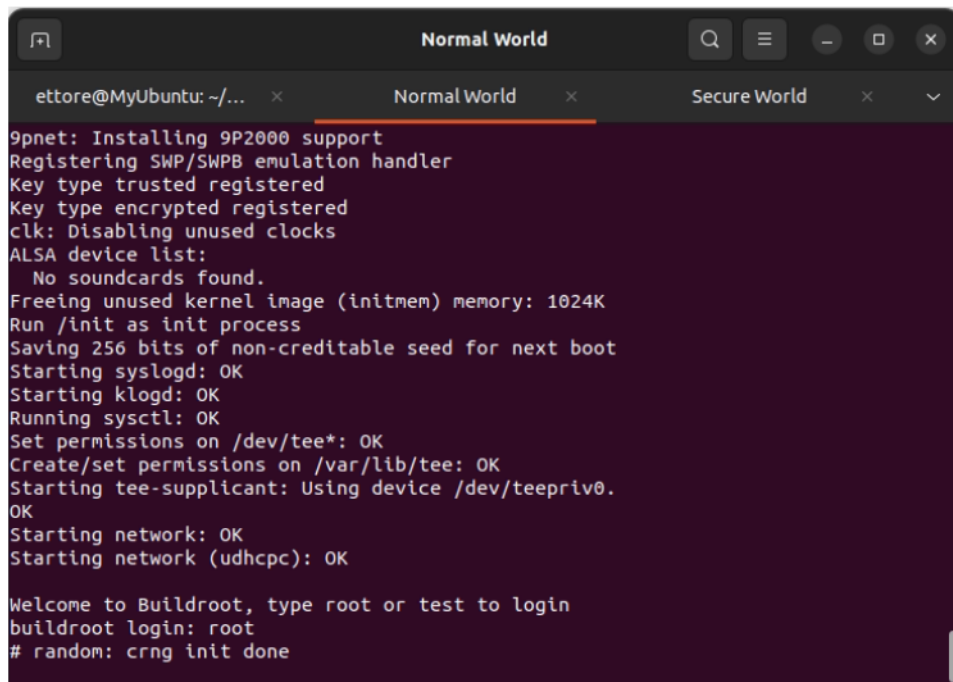
`c`

Instead, to close the program, enter:

`q`

Additional options are available; to explore them, simply enter the *help* command in the QEMU console. This will display a list of all available QEMU commands.

After that, log in by entering either *root* or *test* as a password. At this point, the setup is complete. A short wait is required for the system to load the fTPM into the kernel. Once the process is completed, a confirmation message will appear on the screen:



```

Normal World
ettore@MyUbuntu: ~/... x Normal World x Secure World x v
9pnet: Installing 9P2000 support
Registering SWP/SWPB emulation handler
Key type trusted registered
Key type encrypted registered
clk: Disabling unused clocks
ALSA device list:
  No soundcards found.
Freeing unused kernel image (initmem) memory: 1024K
Run /init as init process
Saving 256 bits of non-creditable seed for next boot
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Set permissions on /dev/tee*: OK
Create/set permissions on /var/lib/tee: OK
Starting tee-supPLICANT: Using device /dev/teepriv0.
OK
Starting network: OK
Starting network (udhcpc): OK

Welcome to Buildroot, type root or test to login
buildroot login: root
# random: crng init done

```

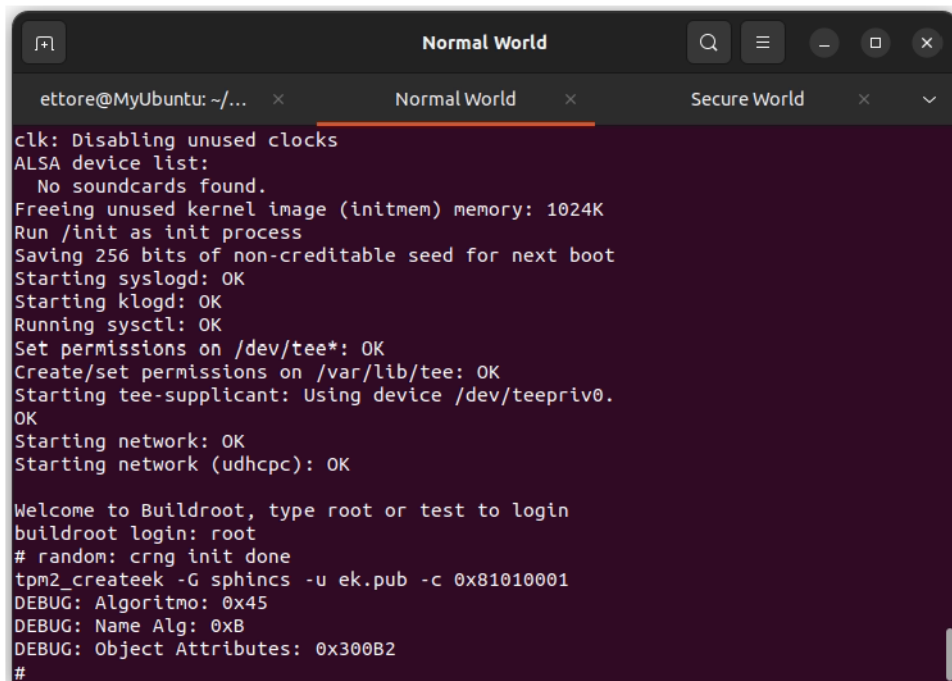
A.3.1 Example of commands

This subsection illustrates an example of a *tpm2-tools* command using the newly integrated algorithm, and how the system handles and displays the response.

To create a SPHINCS Endorsement Key, the *tpm2_createek* command can be executed as follows:

```
tpm2_createek -G sphincs -u ek.pub -c 0x81010001
```

It creates a SPHINCS Endorsement Key and makes it persistent in the TPM. All other commands follow the syntax defined in the official *tpm2-tools* documentation [79]. In Figure A.1, there is a demonstration of the command shown above.

A terminal window titled "Normal World" with a dark purple background and white text. The window has a title bar with a search icon, a menu icon, and window control buttons. Below the title bar, there are three tabs: "ettore@MyUbuntu: ~/...", "Normal World", and "Secure World". The "Normal World" tab is selected. The terminal output shows the following commands and their results:

```
clk: Disabling unused clocks
ALSA device list:
  No soundcards found.
Freeing unused kernel image (initmem) memory: 1024K
Run /init as init process
Saving 256 bits of non-creditable seed for next boot
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Set permissions on /dev/tee*: OK
Create/set permissions on /var/lib/tee: OK
Starting tee-supPLICant: Using device /dev/teepriv0.
OK
Starting network: OK
Starting network (udhcpc): OK

Welcome to Buildroot, type root or test to login
buildroot login: root
# random: crng init done
tpm2_createek -G sphincs -u ek.pub -c 0x81010001
DEBUG: Algoritmo: 0x45
DEBUG: Name Alg: 0xB
DEBUG: Object Attributes: 0x300B2
#
```

Figure A.1: Example of a TPM command

Appendix B

Developer Manual

This developer manual describes the implementation of a firmware TPM (fTPM) running in the Secure World of OP-TEE. This manual is intended for developers who wish to understand the changes made to the system, as well as those who want to replicate or further modify the project.

B.1 Requirements

- OP-TEE ([GitHub OP-TEE](#))
- Liboqs ([GitHub Liboqs](#))
- Linux OS

B.2 Installation

B.2.1 Install Dependencies

Install dependencies on Ubuntu:

```
$ sudo apt install device-tree-compiler bison flex libssl-dev g++  
libgmp-dev libmpc-dev ninja-build pkg-config libglib2.0-dev  
libpixman-1-dev meson cmake gcc python3-pytest python3-pytest-xdist  
unzip xsltproc doxygen graphviz python3-yaml valgrind
```

B.2.2 Cloning of repositories

For the first step, you have to download and install the OP-TEE project

```
$ mkdir optee  
$ cd optee  
$ repo init -u https://github.com/OP-TEE/manifest.git  
$ repo sync
```

Then you have to download the liboqs project

```
$ cd optee  
$ git clone -b main https://github.com/open-quantum-safe/liboqs.git
```

B.3 Project Configuration

B.3.1 Build

The build directory contains all the mk files needed to build the entire project. There are different kinds of files, but the one that needs to be implemented is the `qemu.mk` file. It was added a few lines of code to build also the liboqs project. Now we have to set two new targets inside the `qemu.mk` file:

```
#####
# LIBOQS
#####
oqs:
    if [ ! -d $(LIBOQS_PATH)/build ]; then \
        mkdir -p $(LIBOQS_PATH)/build; \
        cmake -S $(LIBOQS_PATH) -B $(LIBOQS_PATH)/build \
            -DCMAKE_TOOLCHAIN_FILE=$(LIBOQS_PATH)/toolchain-arm.cmake \
            -DCMAKE_C_COMPILER=$(AARCH32_CROSS_COMPILE) \
            -DOQS_PERMIT_UNSUPPORTED_ARCHITECTURE=ON \
            -DOQS_ENABLE_KEM_CLASSIC_MCELIECE:BOOL=OFF \
            -DOQS_USE_OPENSSL=OFF -DOQS_BUILD_ONLY_LIB=ON; \
    fi
    $(MAKE) -C $(LIBOQS_PATH) \
        TA_DEV_KIT_DIR=$(OPTEE_OS_PATH)/out/arm/export-ta_arm32 \
        CROSS_COMPILE=$(AARCH32_CROSS_COMPILE) \
        --no-builtin-variables
    $(MAKE) -C $(LIBOQS_PATH) install \
        TA_DEV_KIT_DIR=$(OPTEE_OS_PATH)/out/arm/export-ta_arm32

oqs-clean:
    $(MAKE) -C $(LIBOQS_PATH) clean \
        TA_DEV_KIT_DIR=$(OPTEE_OS_PATH)/out/arm/export-ta_arm32
    $(MAKE) -C $(LIBOQS_PATH) uninstall \
        TA_DEV_KIT_DIR=$(OPTEE_OS_PATH)/out/arm/export-ta_arm32
```

In this context, special attention should be given to the flags `-DOQS_PERMIT_UNSUPPORTED_ARCHITECTURE` and `-DOQS_USE_OPENSSL`. The former is necessary when building the project on architectures other than ARM, while the latter is required because OP-TEE does not provide OpenSSL, making it necessary to disable OpenSSL in the compilation of liboqs. In this project, OpenSSL is not used; instead, cryptographic primitives from liboqs or standard library functions like `getrandom` are utilized. Additionally, the `-DOQS_BUILD_ONLY_LIB` flag is included solely to accelerate the build process.

They are needed to enable the build of the liboqs project inside the OP-TEE project. Then they must be added in the `make` target:

```
all: oqs arm-tf u-boot buildroot linux optee-os qemu
clean: oqs-clean arm-tf-clean u-boot-clean buildroot-clean linux-clean
      optee-os-clean \
      qemu-clean check-clean
```

B.3.2 Liboqs

This directory contains all the source code needed to implement PQC algorithms. The liboqs source code is taken from [80]. The structure of the liboqs project is the following.



Figure B.1: Directory tree of liboqs project

In the liboqs directory must be added two files: *toolchain-arm-cmake* e *Makefile*. These two files are added to automatize the build procedure. In particular, in the Makefile are defined the *make* and *make install* commands able to start the building procedure. In the *toolchain-arm.cmake* are defined all the flags needed to configure the project. Next, it will be discussed the content of these two files. In Makefile:

```

ROOT      ?= $(CURDIR)
LIBOQS_ROOT ?= $(ROOT)/build

.PHONY: all
all: build

.PHONY: build
build:
    $(MAKE) -C $(LIBOQS_ROOT) \
    PLATFORM=$(PLATFORM) \
    CROSS_COMPILE=$(CROSS_COMPILE) \
    TA_DEV_KIT_DIR=$(TA_DEV_KIT_DIR)

.PHONY: install
install:
    mkdir -p $(TA_DEV_KIT_DIR)/lib
    cp $(LIBOQS_ROOT)/lib/liboqs.a $(TA_DEV_KIT_DIR)/lib/. && \
    mkdir -p $(TA_DEV_KIT_DIR)/include/oqs && \
    cp $(LIBOQS_ROOT)/include/oqs/* $(TA_DEV_KIT_DIR)/include/oqs/.

.PHONY: clean
clean:
    rm -rf build

.PHONY: uninstall
uninstall:
    rm -r $(TA_DEV_KIT_DIR)/include/oqs; \
    rm $(TA_DEV_KIT_DIR)/lib/liboqs.a;

```

The *build* directive builds the oqs project and generates the library named *liboqs.a*. It is a static library needed to add all the post-quantum functions inside it. It is possible also to build the project to create the shared library, but it is not needed for this implementation. Then, the *install* directive is run to transfer the static library inside the OP-TEE project. The last two directives are needed only to clean the project. The other important file is the *toolchain-arm.make*, in which are defined all the flags needed to build liboqs.

In *toolchain-arm.cmake*:

```
# toolchain-arm.cmake
SET(CMAKE_SYSTEM_NAME Linux)
SET(CMAKE_SYSTEM_PROCESSOR arm)

SET(CMAKE_C_COMPILER arm-linux-gnueabi-hf-gcc)
SET(CMAKE_CXX_COMPILER arm-linux-gnueabi-hf-g++)
SET(CMAKE_AR arm-linux-gnueabi-hf-ar)
SET(CMAKE_LINKER arm-linux-gnueabi-hf-ld)

SET(CMAKE_C_FLAGS "-march=armv7-a-mcpu=neon-02-fPIC")
SET(CMAKE_CXX_FLAGS "-march=armv7-a-mcpu=neon-02-fPIC")
SET(CMAKE_EXE_LINKER_FLAGS "-march=armv7-a-mcpu=neon-02-fPIC")
```

It sets the architecture for which it is built and the processor. It is important because, inside the Open Quantum Project, some optimizations depend on the architecture and the operating system of the device.

B.3.3 Optee_os

Now you have to link the liboqs library with optee_os and all the TAs inside OP-TEE. This can be done inserting two lines in *optee_os/ta/mk/build-user-ta.mk* and *optee_os/ta/mk/ta_dev_kit.mk*. In *build-user-ta.mk*:

```
libnames = utils utee oqs
```

In *ta_dev_kit.mk*:

```
libnames += oqs
libdeps += $(ta-dev-kit-dir$(sm))/lib/liboqs.a
```

These lines enable the integration of the static liboqs library within OP-TEE. In this implementation, liboqs has been incorporated separately into both OP-TEE and fTPM to ensure that fTPM functions can execute independently of liboqs. This approach allows liboqs functions to be utilized by all Trusted Applications (TAs) within OP-TEE. Additionally, the static library and its corresponding headers must be located in specific directories: *optee_os/out/arm/export-ta_arm32/lib* for the library *optee_os/out/arm/export-ta_arm32/include* for the headers.

Furthermore, it is necessary to implement certain functions that are not provided by OP-TEE but are required for the integration of liboqs. In this particular implementation, the functions *fprintf_chk*, *memcpy_chk*, *explicit_bzero_chk*, and *assert_fail* were added, although the required functions may vary depending on the target board. These implementations have been placed within the libutils library directory, and the corresponding sub.mk file has been updated accordingly.

B.3.4 Ms-tpm-20-ref

In this directory is contained all the source code needed to implement the fTPM. In order to add new algorithms, first of all it must be registered among the algorithms managed by the TPM. This requires adding two new constants: one to define the algorithm's identifier and another to allow the TPM to recognize and accept it. The first constant is added to *TpmTypes.h*, while the second is included in the *TpmProfile.h* file.

```
/* New Value that identifies the new algorithm */
#define ALG_SPHINCS_SHAKE_256F_VALUE 0x0045
#define TPM_ALG_SPHINCS_SHAKE_256F
      (TPM_ALG_ID)(ALG_SPHINCS_SHAKE_256F_VALUE)

/* Abilitate the TPM to accept the new algorithm */
```

```
#ifndef ALG_LIBOQS
#define ALG_LIBOQS    ALG_YES
#endif
```

Additionally, it is necessary to introduce new structures required for managing the new algorithm. This was achieved by adding the following lines to the *TpmTypes.h* file:

```
typedef union {
    struct {
        UINT16  size;
        BYTE  buffer[ALG_SPHINCS_PUBLIC_KEY_BYTES];
    }  t;
    TPM2B  b;
} TPM2B_PUBLIC_KEY_SPHINCS

typedef union {
    struct {
        UINT16  size;
        BYTE  buffer[ALG_SPHINCS_PRIVATE_KEY_BYTES];
    }  t;
    TPM2B  b;
} TPM2B_PRIVATE_KEY_SPHINCS;

typedef union {
    struct {
        UINT16  size;
        BYTE  *buffer;
    }  t;
    TPM2B  b;
} TPM2B_SIGNATURE_SPHINCS;

typedef struct {
    TPMI_ALG_HASH  hash;
    TPM2B_SIGNATURE_SPHINCS  sig;
} TPMS_SIGNATURE_SPHINCS;
```

These structures follow a common design: a *size* field specifies the length of the key or signature, while a buffer stores the actual key or signature. The key difference to highlight is between the signature structure and two key structures. In the case of the signature, the buffer is not statically allocated because the size of SPHINCS-SHAKE-256f-simple is too large to fit into the stack. Instead, the solution relies on heap memory and dynamic allocation to store the signature, as the heap is generally larger than the stack. However, this approach introduces the typical challenges associated with dynamic memory management in C, requiring careful handling to ensure memory is properly released when no longer needed.

Next, these structures must be integrated into the existing ones responsible for handling public keys, private keys, and digital signatures to support the new algorithm:

```
typedef union {
    #if ALG_KEYEDHASH
        TPM2B_DIGEST  keyedHash;
    #endif // ALG_KEYEDHASH
    #if ALG_SYMCIPHER
        TPM2B_DIGEST  sym;
    #endif // ALG_SYMCIPHER
    #if ALG_RSA
        TPM2B_PUBLIC_KEY_RSA  rsa;
    #endif // ALG_RSA
    #if ALG_ECC
```

```
    TPMS_ECC_POINT ecc;
#endif // ALG_ECC
#if ALG_LIBOQS
    TPM2B_PUBLIC_KEY_SPHINCS sphincs;
#endif // ALG_LIBOQS
    TPMS_DERIVE derive;
} TPMU_PUBLIC_ID;

typedef union {
#if ALG_RSA
    TPM2B_PRIVATE_KEY_RSA rsa;
#endif // ALG_RSA
#if ALG_ECC
    TPM2B_ECC_PARAMETER ecc;
#endif // ALG_ECC
#if ALG_KEYEDHASH
    TPM2B_SENSITIVE_DATA bits;
#endif // ALG_KEYEDHASH
#if ALG_SYMCIPHER
    TPM2B_SYM_KEY sym;
#endif // ALG_SYMCIPHER
#if ALG_LIBOQS
    TPM2B_PRIVATE_KEY_SPHINCS sphincs;
#endif // ALG_LIBOQS
    TPM2B_PRIVATE_VENDOR_SPECIFIC any;
} TPMU_SENSITIVE_COMPOSITE;

typedef union {
#if ALG_ECC
    TPMS_SIGNATURE_ECDAE ecdaa;
#endif // ALG_ECC
#if ALG_RSA
    TPMS_SIGNATURE_RSASSA rsassa;
#endif // ALG_RSA
#if ALG_RSA
    TPMS_SIGNATURE_RSAPSS rsapss;
#endif // ALG_RSA
#if ALG_ECC
    TPMS_SIGNATURE_ECDSA ecdsa;
#endif // ALG_ECC
#if ALG_ECC
    TPMS_SIGNATURE_SM2 sm2;
#endif // ALG_ECC
#if ALG_ECC
    TPMS_SIGNATURE_ECSCHNORR ecschnorr;
#endif // ALG_ECC
#if ALG_HMAC
    TPMT_HA hmac;
#endif // ALG_HMAC
#if ALG_LIBOQS
    TPMS_SIGNATURE_SPHINCS sphincs;
#endif
    TPMS_SCHEME_HASH any;
} TPMU_SIGNATURE;
```

A new folder named *liboqs* has been added to the implementation, containing all the necessary source files from the liboqs library for key generation and SPHINCS signature operations. This approach allows direct integration of the required functions within fTPM, avoiding reliance on an

external library while ensuring that modifications affect only fTPM and not the entire OP-TEE system.

This directory is located in `TPMcmd/tpm/src/crypt` to maintain consistency by keeping all cryptographic-related files in the same location.

To support SPHINCS-SHAKE-256f-simple, a new file named *CryptLiboqs.c* has been added to the ms-tpm-20-ref source code. This file utilizes the newly integrated liboqs functions to handle key generation, signature creation, and signing operations within the TPM. Additionally, a corresponding header file has been included to properly structure and expose the required functionalities within the implementation.

```
#include "Tpm.h"
#include "CryptLiboqs.h"
#include "nistapi.h"

#define OQS_SIG_sphincs_shake_256f_simple_length_public_key 64
#define OQS_SIG_sphincs_shake_256f_simple_length_secret_key 128
#define OQS_SIG_sphincs_shake_256f_simple_length_signature 49856

TPM_RC CryptSphincsGenerateKeyPair(
    TPMT_PUBLIC *publicArea,
    TPMT_SENSITIVE *sensitive,
    RAND_STATE *rand){

    uint8_t
        publicKey[OQS_SIG_sphincs_shake_256f_simple_length_public_key];
    uint8_t
        privateKey[OQS_SIG_sphincs_shake_256f_simple_length_secret_key];

    if (crypto_sign_keypair(publicKey, privateKey) != OQS_SUCCESS)
    {
        return TPM_RC_FAILURE;
    }

    publicArea->unique.sphincs.t.size =
        OQS_SIG_sphincs_shake_256s_simple_length_public_key;
    memcpy(publicArea->unique.sphincs.t.buffer, publicKey,
        OQS_SIG_sphincs_shake_256s_simple_length_public_key);

    sensitive->sensitive.sphincs.t.size =
        OQS_SIG_sphincs_shake_256s_simple_length_secret_key;
    memcpy(sensitive->sensitive.sphincs.t.buffer, privateKey,
        OQS_SIG_sphincs_shake_256s_simple_length_secret_key);

    return TPM_RC_SUCCESS;
}

TPM_RC CryptSphincsSign(TPMT_SIGNATURE* sigOut,
    OBJECT* key,
    TPM2B_DIGEST* digest,
    RAND_STATE* rand
){

    TPM_RC retVal = TPM_RC_SUCCESS;
    UINT16 modSize;
```

```

pAssert(sigOut != NULL && key != NULL && digest !=
        NULL);

/* Dynamically allocate the signature buffer */
sigOut->signature.sphincs.sig.t.buffer =
    malloc(OQS_SIG_sphincs_shake_256f_simple_length_signature);

modSize = key->publicArea.unique.sphincs.t.size;

if(retVal == TPM_RC_SUCCESS){

    if(crypto_sign_signature(
        sigOut->signature.sphincs.sig.t.buffer/*
        Signature buffer */,
        &sigOut->signature.sphincs.sig.t.size /*
        Signature size */,
        digest->b.buffer /* Signing message */,
        digest->b.size /* Message size */,
        key->sensitive.sensitive.sphincs.t.buffer
        /* secret key*/) != OQS_SUCCESS)
        return TPM_RC_FAILURE;
    free(sigOut->signature.sphincs.sig.t.buffer);
}

return TPM_RC_FAILURE;
}

```

These two new functions are added to handle the generation of the SPHINCS+ key or SPHINCS+ signature respectively. These new functions use the liboqs primitives to create a key or sign a message. Then, these functions are inserted into existing TPM functions, to extend their functionalities and to include the new algorithm. These functions are inserted in this way:

```

#if ALG_LIBOQS
    // Create SPHINCS+ key
    case TPM_ALG_SPHINCS_SHAKE_256F:
        result = CryptSphincsGenerateKeyPair(publicArea, sensitive, rand);
        break;
#endif // ALG_LIBOQS

```

This line is inserted inside `CryptCreateObject` function. It is added in a switch case in order to generate a SPHINCS keypair. If the field *type* of `publicArea` is the new algorithm, the TPM runs the *CryptSphincsGenerateKeyPair* that generates the new SPHINCS+ keypair and stores them into the appropriate structures.

```

#if ALG_LIBOQS
    case TPM_ALG_SPHINCS_SHAKE_256F:
        result = CryptSphincsSign(signature, signKey, digest, NULL);
        break;
#endif //ALG_LIBOQS

```

This line is added inside *CryptSign* function in *CryptUtil.c* file. It only provides the possibility to sign with SPHINCS. In this case, because of the signature length, the signature is not stored inside the TPM otherwise running out of space inside the TPM and would be a real problem. Additionally, in order to be able to move data from one world to another, it is essential to tell how the data should be serialized or deserialized. This is the scope of Marshalling and Unmarshalling functions, which have been extended to include new added structures. This is done within the fTPM support folder, specifically within the *Marshal.c* file. The following lines of code have been added to it:

```

#if ALG_LIBOQS
TPM_RC
TPM2B_PUBLIC_KEY_SPHINCS_Unmarshal(TPM2B_PUBLIC_KEY_SPHINCS *target, BYTE
    **buffer, INT32 *size)
{
    TPM_RC result;
    result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
    if(result == TPM_RC_SUCCESS)
    {
        if((target->t.size) > 64)
            result = TPM_RC_SIZE;
        else
            result = BYTE_Array_Unmarshal((BYTE *)&(target->t.buffer), buffer,
                size, (INT32)(target->t.size));
    }
    return result;
}

UINT16
TPM2B_PUBLIC_KEY_SPHINCS_Marshal(TPM2B_PUBLIC_KEY_SPHINCS *source, BYTE
    **buffer, INT32 *size)
{
    UINT16 result = 0;
    result = (UINT16)(result + UINT16_Marshal((UINT16 *)&(source->t.size),
        buffer, size));
    // if size equal to 0, the rest of the structure is a zero buffer. Stop
    // processing
    if(source->t.size == 0)
        return result;
    result = (UINT16)(result + BYTE_Array_Marshal((BYTE *)&(source->t.buffer),
        buffer, size, (INT32)(source->t.size)));
    return result;
}
#endif // ALG_LIBOQS

#if ALG_LIBOQS
TPM_RC
TPM2B_SIGNATURE_SPHINCS_Unmarshal(TPM2B_SIGNATURE_SPHINCS *target, BYTE
    **buffer, INT32 *size)
{
    TPM_RC result;
    result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
    if(result == TPM_RC_SUCCESS)
    {
        if((target->t.size) > 64)
            result = TPM_RC_SIZE;
        else
            result = BYTE_Array_Unmarshal((BYTE *)&(target->t.buffer), buffer,
                size, (INT32)(target->t.size));
    }
    return result;
}

UINT16
TPM2B_SIGNATURE_SPHINCS_Marshal(TPM2B_SIGNATURE_SPHINCS *source, BYTE
    **buffer, INT32 *size)
{

```

```

    UINT16 result = 0;
    result = (UINT16)(result + UINT16_Marshal((UINT16 *)&(source->t.size),
        buffer, size));
    // if size equal to 0, the rest of the structure is a zero buffer. Stop
    processing
    if(source->t.size == 0)
        return result;
    result = (UINT16)(result + BYTE_Array_Marshal((BYTE *)&(source->t.buffer),
        buffer, size, (INT32)(source->t.size)));
    return result;
}

#endif // ALG_LIBOQS

#if ALG_LIBOQS
TPM_RC
TPM2B_PRIVATE_KEY_SPHINCS_Unmarshal(TPM2B_PRIVATE_KEY_SPHINCS *target, BYTE
    **buffer, INT32 *size)
{
    TPM_RC result;
    result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
    if(result == TPM_RC_SUCCESS)
    {
        if((target->t.size) > 128)
            result = TPM_RC_SIZE;
        else
            result = BYTE_Array_Unmarshal((BYTE *)&(target->t.buffer), buffer,
                size, (INT32)(target->t.size));
    }
    return result;
}

UINT16
TPM2B_PRIVATE_KEY_SPHINCS_Marshal(TPM2B_PRIVATE_KEY_SPHINCS *source, BYTE
    **buffer, INT32 *size)
{
    UINT16 result = 0;
    result = (UINT16)(result + UINT16_Marshal((UINT16 *)&(source->t.size),
        buffer, size));
    // if size equal to 0, the rest of the structure is a zero buffer. Stop
    processing
    if(source->t.size == 0)
        return result;
    result = (UINT16)(result + BYTE_Array_Marshal((BYTE *)&(source->t.buffer),
        buffer, size, (INT32)(source->t.size)));
    return result;
}

#endif //ALG_LIBOQS

#if ALG_LIBOQS
TPM_RC
TPMS_SIGNATURE_SPHINCS_Unmarshal(TPMS_SIGNATURE_SPHINCS *target, BYTE
    **buffer, INT32 *size)
{
    TPM_RC result;
    result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)&(target->hash),
        buffer, size, 0);
}

```



```

    if(result == TPM_RC_SUCCESS)
        result = TPM2B_SIGNATURE_SPHINCS_Unmarshal((TPM2B_SIGNATURE_SPHINCS
            *)&(target->sig), buffer, size);
    return result;
}

UINT16
TPMS_SIGNATURE_SPHINCS_Marshal(TPMS_SIGNATURE_SPHINCS *source, BYTE **buffer,
    INT32 *size)
{
    UINT16 result = 0;
    result = (UINT16)(result + TPMI_ALG_HASH_Marshal((TPMI_ALG_HASH
        *)&(source->hash), buffer, size));
    result = (UINT16)(result +
        TPM2B_SIGNATURE_SPHINCS_Marshal((TPM2B_SIGNATURE_SPHINCS
            *)&(source->sig), buffer, size));
    return result;
}
#endif // ALG_LIBOQS

TPM_RC
TPMU_SIGNATURE_Unmarshal(TPMU_SIGNATURE *target, BYTE **buffer, INT32 *size,
    UINT32 selector)
{
    switch(selector) {
    #if ALG_ECDSA
        case TPM_ALG_ECDSA:
            return TPMS_SIGNATURE_ECDSA_Unmarshal((TPMS_SIGNATURE_ECDSA
                *)&(target->ecdsa), buffer, size);
    #endif // ALG_ECDSA
    #if ALG_RSASSA
        case TPM_ALG_RSASSA:
            return TPMS_SIGNATURE_RSASSA_Unmarshal((TPMS_SIGNATURE_RSASSA
                *)&(target->rsassa), buffer, size);
    #endif // ALG_RSASSA
    #if ALG_RSAPSS
        case TPM_ALG_RSAPSS:
            return TPMS_SIGNATURE_RSAPSS_Unmarshal((TPMS_SIGNATURE_RSAPSS
                *)&(target->rsapss), buffer, size);
    #endif // ALG_RSAPSS
    #if ALG_ECDSA
        case TPM_ALG_ECDSA:
            return TPMS_SIGNATURE_ECDSA_Unmarshal((TPMS_SIGNATURE_ECDSA
                *)&(target->ecdsa), buffer, size);
    #endif // ALG_ECDSA
    #if ALG_SM2
        case TPM_ALG_SM2:
            return TPMS_SIGNATURE_SM2_Unmarshal((TPMS_SIGNATURE_SM2
                *)&(target->sm2), buffer, size);
    #endif // ALG_SM2
    #if ALG_ECSCHNORR
        case TPM_ALG_ECSCHNORR:
            return
                TPMS_SIGNATURE_ECSCHNORR_Unmarshal((TPMS_SIGNATURE_ECSCHNORR
                    *)&(target->ecschnorr), buffer, size);
    #endif // ALG_ECSCHNORR
    #if ALG_HMAC

```

```

        case TPM_ALG_HMAC:
            return TPMT_HA_Unmarshal((TPMT_HA *)&(target->hmac), buffer, size,
                                     0);
    #endif // ALG_HMAC
    #if ALG_LIBOQS
        case TPM_ALG_SPHINCS_SHAKE_256F:
            return TPMS_SIGNATURE_SPHINCS_Unmarshal((TPMS_SIGNATURE_SPHINCS
                                                       *)&(target->sphincs), buffer, size);
    #endif // ALG_LIBOQS
        case TPM_ALG_NULL:
            return TPM_RC_SUCCESS;
    }
    return TPM_RC_SELECTOR;
}

UINT16
TPMU_SIGNATURE_Marshal(TPMU_SIGNATURE *source, BYTE **buffer, INT32 *size,
                       UINT32 selector)
{
    switch(selector) {
    #if ALG_ECDSA
        case TPM_ALG_ECDSA:
            return TPMS_SIGNATURE_ECDSA_Marshal((TPMS_SIGNATURE_ECDSA
                                                  *)&(source->ecdsa), buffer, size);
    #endif // ALG_ECDSA
    #if ALG_RSASSA
        case TPM_ALG_RSASSA:
            return TPMS_SIGNATURE_RSASSA_Marshal((TPMS_SIGNATURE_RSASSA
                                                  *)&(source->rsassa), buffer, size);
    #endif // ALG_RSASSA
    #if ALG_RSAPSS
        case TPM_ALG_RSAPSS:
            return TPMS_SIGNATURE_RSAPSS_Marshal((TPMS_SIGNATURE_RSAPSS
                                                  *)&(source->rsapss), buffer, size);
    #endif // ALG_RSAPSS
    #if ALG_ECDSA
        case TPM_ALG_ECDSA:
            return TPMS_SIGNATURE_ECDSA_Marshal((TPMS_SIGNATURE_ECDSA
                                                  *)&(source->ecdsa), buffer, size);
    #endif // ALG_ECDSA
    #if ALG_SM2
        case TPM_ALG_SM2:
            return TPMS_SIGNATURE_SM2_Marshal((TPMS_SIGNATURE_SM2
                                               *)&(source->sm2), buffer, size);
    #endif // ALG_SM2
    #if ALG_ECSCHNORR
        case TPM_ALG_ECSCHNORR:
            return TPMS_SIGNATURE_ECSCHNORR_Marshal((TPMS_SIGNATURE_ECSCHNORR
                                                      *)&(source->ecschnorr), buffer, size);
    #endif // ALG_ECSCHNORR
    #if ALG_HMAC
        case TPM_ALG_HMAC:
            return TPMT_HA_Marshal((TPMT_HA *)&(source->hmac), buffer, size);
    #endif // ALG_HMAC
    #if ALG_LIBOQS
        case TPM_ALG_SPHINCS_SHAKE_256F:

```

```

        return TPMS_SIGNATURE_SPHINCS_Marshal((TPMS_SIGNATURE_SPHINCS
            *)&(source->sphincs), buffer, size);
#endif // ALG_LIBOQS
        case TPM_ALG_NULL:
            return 0;
        }
        return 0;
    }

TPM_RC
TPMU_PUBLIC_ID_Unmarshal(TPMU_PUBLIC_ID *target, BYTE **buffer, INT32 *size,
    UINT32 selector)
{
    switch(selector) {
#if ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST
                *)&(target->keyedHash), buffer, size);
#endif // ALG_KEYEDHASH
#if ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)&(target->sym),
                buffer, size);
#endif // ALG_SYMCIPHER
#if ALG_RSA
        case TPM_ALG_RSA:
            return TPM2B_PUBLIC_KEY_RSA_Unmarshal((TPM2B_PUBLIC_KEY_RSA
                *)&(target->rsa), buffer, size);
#endif // ALG_RSA
#if ALG_LIBOQS
        case TPM_ALG_SPHINCS_SHAKE_256F:
            return
                TPM2B_PUBLIC_KEY_SPHINCS_Unmarshal((TPM2B_PUBLIC_KEY_SPHINCS
                    *)&(target->sphincs), buffer, size);
#endif //ALG_LIBOQS
#if ALG_ECC
        case TPM_ALG_ECC:
            return TPMS_ECC_POINT_Unmarshal((TPMS_ECC_POINT *)&(target->ecc),
                buffer, size);
#endif // ALG_ECC
    }
    return TPM_RC_SELECTOR;
}

UINT16
TPMU_PUBLIC_ID_Marshal(TPMU_PUBLIC_ID *source, BYTE **buffer, INT32 *size,
    UINT32 selector)
{
    switch(selector) {
#if ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPM2B_DIGEST_Marshal((TPM2B_DIGEST *)&(source->keyedHash),
                buffer, size);
#endif // ALG_KEYEDHASH
#if ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:

```

```

        return TPM2B_DIGEST_Marshal((TPM2B_DIGEST *)&(source->sym),
        buffer, size);
#endif // ALG_SYMCIPHER
#if ALG_RSA
    case TPM_ALG_RSA:
        return TPM2B_PUBLIC_KEY_RSA_Marshal((TPM2B_PUBLIC_KEY_RSA
        *)&(source->rsa), buffer, size);
#endif // ALG_RSA
#if ALG_LIBOQS
    case TPM_ALG_SPHINCS_SHAKE_256F:
        return TPM2B_PUBLIC_KEY_SPHINCS_Marshal((TPM2B_PUBLIC_KEY_SPHINCS
        *)&(source->sphincs), buffer, size);
#endif //ALG_LIBOQS
#if ALG_ECC
    case TPM_ALG_ECC:
        return TPMS_ECC_POINT_Marshal((TPMS_ECC_POINT *)&(source->ecc),
        buffer, size);
#endif // ALG_ECC
    }
    return 0;
}

TPM_RC
TPMU_SENSITIVE_COMPOSITE_Unmarshal(TPMU_SENSITIVE_COMPOSITE *target, BYTE
    **buffer, INT32 *size, UINT32 selector)
{
    switch(selector) {
#if ALG_RSA
        case TPM_ALG_RSA:
            return TPM2B_PRIVATE_KEY_RSA_Unmarshal((TPM2B_PRIVATE_KEY_RSA
            *)&(target->rsa), buffer, size);
#endif // ALG_RSA
#if ALG_LIBOQS
        case TPM_ALG_SPHINCS_SHAKE_256F:
            return
                TPM2B_PRIVATE_KEY_SPHINCS_Unmarshal((TPM2B_PRIVATE_KEY_SPHINCS
                *)&(target->sphincs), buffer, size);
#endif ALG_LIBOQS
#if ALG_ECC
        case TPM_ALG_ECC:
            return TPM2B_ECC_PARAMETER_Unmarshal((TPM2B_ECC_PARAMETER
            *)&(target->ecc), buffer, size);
#endif // ALG_ECC
#if ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPM2B_SENSITIVE_DATA_Unmarshal((TPM2B_SENSITIVE_DATA
            *)&(target->bits), buffer, size);
#endif // ALG_KEYEDHASH
#if ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPM2B_SYM_KEY_Unmarshal((TPM2B_SYM_KEY *)&(target->sym),
            buffer, size);
#endif // ALG_SYMCIPHER
    }
    return TPM_RC_SELECTOR;
}

```



```

UINT16
TPMU_SENSITIVE_COMPOSITE_Marshal(TPMU_SENSITIVE_COMPOSITE *source, BYTE
    **buffer, INT32 *size, UINT32 selector)
{
    switch(selector) {
    #if ALG_RSA
        case TPM_ALG_RSA:
            return TPM2B_PRIVATE_KEY_RSA_Marshal((TPM2B_PRIVATE_KEY_RSA
                *)&(source->rsa), buffer, size);
    #endif // ALG_RSA
    #if ALG_LIBOQS
        case TPM_ALG_SPHINCS_SHAKE_256F:
            return
                TPM2B_PRIVATE_KEY_SPHINCS_Marshal((TPM2B_PRIVATE_KEY_SPHINCS
                    *)&(source->sphincs), buffer, size);
    #endif ALG_LIBOQS
    #if ALG_ECC
        case TPM_ALG_ECC:
            return TPM2B_ECC_PARAMETER_Marshal((TPM2B_ECC_PARAMETER
                *)&(source->ecc), buffer, size);
    #endif // ALG_ECC
    #if ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPM2B_SENSITIVE_DATA_Marshal((TPM2B_SENSITIVE_DATA
                *)&(source->bits), buffer, size);
    #endif // ALG_KEYEDHASH
    #if ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPM2B_SYM_KEY_Marshal((TPM2B_SYM_KEY *)&(source->sym),
                buffer, size);
    #endif // ALG_SYMCIPHER
    }
    return 0;
}

TPM_RC
TPMU_PUBLIC_PARMS_Unmarshal(TPMU_PUBLIC_PARMS *target, BYTE **buffer, INT32
    *size, UINT32 selector)
{
    switch(selector) {
    #if ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPMS_KEYEDHASH_PARMS_Unmarshal((TPMS_KEYEDHASH_PARMS
                *)&(target->keyedHashDetail), buffer, size);
    #endif // ALG_KEYEDHASH
    #if ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPMS_SYMCIPHER_PARMS_Unmarshal((TPMS_SYMCIPHER_PARMS
                *)&(target->symDetail), buffer, size);
    #endif // ALG_SYMCIPHER
    #if ALG_RSA
        case TPM_ALG_RSA:
            return TPMS_RSA_PARMS_Unmarshal((TPMS_RSA_PARMS
                *)&(target->rsaDetail), buffer, size);
    #endif // ALG_RSA
    #if ALG_LIBOQS
        case TPM_ALG_SPHINCS_SHAKE_256F:

```

```

        return TPMS_RSA_PARMS_Unmarshal((TPMS_RSA_PARMS
            *)&(target->rsaDetail), buffer, size);
#endif //ALG_LIBOQS
#if ALG_ECC
    case TPM_ALG_ECC:
        return TPMS_ECC_PARMS_Unmarshal((TPMS_ECC_PARMS
            *)&(target->eccDetail), buffer, size);
#endif // ALG_ECC
    }
    return TPM_RC_SELECTOR;
}
UINT16
TPMU_PUBLIC_PARMS_Marshal(TPMU_PUBLIC_PARMS *source, BYTE **buffer, INT32
    *size, UINT32 selector)
{
    switch(selector) {
#if ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPMS_KEYEDHASH_PARMS_Marshal((TPMS_KEYEDHASH_PARMS
                *)&(source->keyedHashDetail), buffer, size);
#endif // ALG_KEYEDHASH
#if ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPMS_SYMCIPHER_PARMS_Marshal((TPMS_SYMCIPHER_PARMS
                *)&(source->symDetail), buffer, size);
#endif // ALG_SYMCIPHER
#if ALG_RSA
        case TPM_ALG_RSA:
            return TPMS_RSA_PARMS_Marshal((TPMS_RSA_PARMS
                *)&(source->rsaDetail), buffer, size);
#endif // ALG_RSA
#if ALG_LIBOQS
        case TPM_ALG_SPHINCS_SHAKE_256F:
            return TPMS_RSA_PARMS_Marshal((TPMS_RSA_PARMS
                *)&(source->rsaDetail), buffer, size);
#endif ALG_LIBOQS
#if ALG_ECC
        case TPM_ALG_ECC:
            return TPMS_ECC_PARMS_Marshal((TPMS_ECC_PARMS
                *)&(source->eccDetail), buffer, size);
#endif // ALG_ECC
    }
    return 0;
}

```

These functions are crucial to ensure seamless data transmission between components. Without them, errors may occur on one end, depending on the direction of data flow, as the receiving side may fail to recognize the transmitted data.

B.3.5 Optee_ftpm

The *optee_ftpm* module contains the Trusted Application (TA) that must run within the Secure World in OP-TEE. Its source code derived from the *ms-tpm-20-ref* folder. Since additional files have been introduced into the source code, they must be included in the TA's *sub.mk* file to ensure they are compiled during the build process. The lines to be added are as follows:

```
global-incdirs_ext-y += $(CFG_MS_TPM_20_REF)/TPMCmd/tpm/src/crypt/liboqs
```

```
srcs_ext-y += crypt/CryptLiboqs.c

srcs_ext-y += crypt/liboqs/address.c
srcs_ext-y += crypt/liboqs/context_shake.c
srcs_ext-y += crypt/liboqs/fors.c
srcs_ext-y += crypt/liboqs/hash_shake.c
srcs_ext-y += crypt/liboqs/merkle.c
srcs_ext-y += crypt/liboqs/sign.c
srcs_ext-y += crypt/liboqs/thash_shake_simple.c
srcs_ext-y += crypt/liboqs/utils.c
srcs_ext-y += crypt/liboqs/utillsx1.c
srcs_ext-y += crypt/liboqs/wots.c
srcs_ext-y += crypt/liboqs/wotsx1.c
srcs_ext-y += crypt/liboqs/fips202.c
srcs_ext-y += crypt/liboqs/fips202x4.c
```

If these files are not added in this way, they will not be compiled and therefore the changes made in these files will not be executed by fTPM.

B.3.6 Tpm2-tss and Tpm2-tools

To build tpm2-tools and tpm2-tss, the Buildroot build system was used as a foundation. However, in order to tailor the build process to the specific needs of the project, some manual changes were made. Specifically, the .mk files located in the tpm2-tools and tpm2-tss package directories within the Buildroot tree were edited. These adjustments made it possible to customize configuration options, handle dependencies more appropriately, and apply any required patches. The lines required are the following:

```
TPM2_TOOLS_SITE = /home/ettore/optee_linaro/tpm2-tools
TPM2_TOOLS_SITE_METHOD = local

TPM2_TSS_SITE = /home/ettore/optee_linaro/tpm2-tss
TPM2_TSS_SITE_METHOD = local
```

The TPM2_TSS_SITE and TPM2_TOOLS_SITE variables must be set to the path where the new or custom implementation of the respective packages is located. This ensures that Buildroot pulls the correct source code during the build process.

Another significant modification involved adding new data structures within tpm2-tss, necessary to support these new algorithms. Four new structures were introduced and added inside the *tss_tpm2_types.h* file: one for the public key, one for the private key, and two for the signature.

```
#define TPM2_ALG_SPHINCS_SHAKE_256F ((TPM2_ALG_ID) 0x0045)

/* Definition of SPHINCS TPM2B_PUBLIC_KEY_SPHINCS Structure*/
typedef struct TPM2B_PUBLIC_KEY_SPHINCS TPM2B_PRIVATE_KEY_SPHINCS;
struct TPM2B_PUBLIC_KEY_SPHINCS {
    UINT16 size;
    BYTE buffer[TPM2_SPHINCS_PUBLIC_KEY_BITS];
}

/* Definition of SPHINCS TPM2B_PRIVATE_KEY_SPHINCS Structure */
typedef struct TPM2B_PRIVATE_KEY_SPHINCS TPM2B_PRIVATE_KEY_SPHINCS;
struct TPM2B_PRIVATE_KEY_SPHINCS {
    UINT16 size;
    BYTE buffer[TPM2_SPHINCS_SECRET_KEY_BITS];
}
```

```

/* Definition of SPHINCS TPM2B_SIGNATURE_SPHINCS Structure */
typedef struct TPM2B_SIGNATURE_SPHINCS TPM2B_SIGNATURE_SPHINCS;
struct TPM2B_SIGNATURE_SPHINCS {
    UINT16 size;
    BYTE buffer[TPM2_SPHINCS_SIGNATURE_BITS];
}

/* Definition of SPHINCS TPMS_SIGNATURE_SPHINCS Structure */
typedef struct TPMS_SIGNATURE_SPHINCS TPMS_SIGNATURE_SPHINCS;
struct TPMS_SIGNATURE_SPHINCS {
    TPMI_ALG_HASH hash;
    TPM2B_SIGNATURE_SPHINCS sig;
}

```

All the structures follow a common design: they include a *size* field, which indicates the length of the corresponding key or signature, and a buffer that stores the actual key or signature. These structures are modeled similarly to RSA and ECC keys and signatures to maintain compatibility with existing TPM keys. The *TPMS_SIGNATURE_SPHINCS* structure, although it may seem redundant, has been preserved to ensure compatibility with current algorithms and to facilitate future developments. Additionally, these four structures have been integrated into existing structures such as *TPMU_PUBLIC_ID*, *TPMU_SENSITIVE_COMPOSITE*, and *TPMU_SIGNATURE*, enabling SPHINCS keys to coexist with other supported algorithms.

```

/* Definition of TPMU_PUBLIC_ID Union <INOUT S> */
typedef union TPMU_PUBLIC_ID TPMU_PUBLIC_ID;
union TPMU_PUBLIC_ID {
    TPM2B_DIGEST keyedHash;
    TPM2B_DIGEST sym;
    TPM2B_PUBLIC_KEY_RSA rsa;
    TPMS_ECC_POINT ecc;
    TPM2B_PUBLIC_KEY_SPHINCS sphincs;
};

/* Definition of TPMU_SENSITIVE_COMPOSITE Union <INOUT S> */
typedef union TPMU_SENSITIVE_COMPOSITE TPMU_SENSITIVE_COMPOSITE;
union TPMU_SENSITIVE_COMPOSITE {
    TPM2B_PRIVATE_KEY_RSA rsa; /* a prime factor of the public key */
    TPM2B_ECC_PARAMETER ecc; /* the integer private key */
    TPM2B_PRIVATE_KEY_SPHINCS sphincs; /* the sphincs private key */
    TPM2B_SENSITIVE_DATA bits; /* the private data */
    TPM2B_SYM_KEY sym; /* the symmetric key */
    TPM2B_PRIVATE_VENDOR_SPECIFIC any; /* vendor-specific size for key
    storage */
};

/* Definition of TPMU_SIGNATURE Union <INOUT S> */
typedef union TPMU_SIGNATURE TPMU_SIGNATURE;
union TPMU_SIGNATURE {
    TPMS_SIGNATURE_RSASSA rsassa; /* all asymmetric signatures */
    TPMS_SIGNATURE_RSAPSS rsapss; /* all asymmetric signatures */
    TPMS_SIGNATURE_ECDSA ecdsa; /* all asymmetric signatures */
    TPMS_SIGNATURE_ECDSA ecdaa; /* all asymmetric signatures */
    TPMS_SIGNATURE_SM2 sm2; /* all asymmetric signatures */
    TPMS_SIGNATURE_ECSCHNORR ecschnorr; /* all asymmetric signatures */
    TPMS_SIGNATURE_SPHINCS sphincs;
    TPMT_HA hmac; /* HMAC signature required to be supported */
}

```



```

    TPMS_SCHEME_HASH any; /* used to access the hash */
};

```

These structures efficiently handle specific parameters of SPHINCS+ within the TPM architecture, ensuring proper storage and manipulation of the generated keys and signatures. Additionally, a new constant was added to identify the new algorithm, with a hexadecimal value of 0x45. This constant was appended to the long list of supported algorithms and serves to identify the new algorithm and to check whether the algorithm is recognized and usable by the system.

Another fundamental aspect concerns the Marshalling and Unmarshalling functions in tpm2-tss. In this case, some existing functions were modified, and new ones were added to ensure compatibility and proper data transfer. These functions are essential for the structured conversion of information between the various TPM components, ensuring that post-quantum keys and signatures are correctly serialized and deserialized during key generation and signing operations. For this purpose, these lines of code have been added:

```

TPM2B_UNMARSHAL(TPM2B_PUBLIC_KEY_SPHINCS, buffer);
TPM2B_MARSHAL (TPM2B_PRIVATE_KEY_RSA);
TPM2B_UNMARSHAL(TPM2B_PRIVATE_KEY_RSA, buffer);
TPM2B_MARSHAL (TPM2B_SIGNATURE_SPHINCS);
TPM2B_UNMARSHAL(TPM2B_SIGNATURE_SPHINCS, buffer);
TPM2B_MARSHAL (TPM2B_PRIVATE_KEY_SPHINCS);
TPM2B_UNMARSHAL(TPM2B_PRIVATE_KEY_SPHINCS, buffer);

TPMS_MARSHAL_2(TPMS_SIGNATURE_SPHINCS,
               hash, VAL, Tss2_MU_UINT16_Marshal,
               sig, ADDR, Tss2_MU_TPMS_SIGNATURE_SPHINCS_Marshal)

TPMS_UNMARSHAL_2(TPMS_SIGNATURE_SPHINCS,
                 hash, Tss2_MU_UINT16_Unmarshal,
                 sig, Tss2_MU_TPMS_SIGNATURE_SPHINCS_Unmarshal)

TPMU_UNMARSHAL2(TPMU_SIGNATURE,
TPM2_ALG_RSASSA, rsassa, Tss2_MU_TPMS_SIGNATURE_RSA_Unmarshal,
TPM2_ALG_RSAPSS, rsapss, Tss2_MU_TPMS_SIGNATURE_RSA_Unmarshal,
TPM2_ALG_ECDSA, ecdsa, Tss2_MU_TPMS_SIGNATURE_ECC_Unmarshal,
TPM2_ALG_ECDAA, ecdaa, Tss2_MU_TPMS_SIGNATURE_ECC_Unmarshal,
TPM2_ALG_SM2, sm2, Tss2_MU_TPMS_SIGNATURE_ECC_Unmarshal,
TPM2_ALG_ECSCHNORR, ecschnorr, Tss2_MU_TPMS_SIGNATURE_ECC_Unmarshal,
TPM2_ALG_HMAC, hmac, Tss2_MU_TPMT_HA_Unmarshal,
TPM2_ALG_SPHINCS_SHAKE_256F, sphincs,
    Tss2_MU_TPMS_SIGNATURE_SPHINCS_Unmarshal)

TPMU_MARSHAL2(TPMU_SIGNATURE,
TPM2_ALG_RSASSA, ADDR, rsassa, Tss2_MU_TPMS_SIGNATURE_RSA_Marshal,
TPM2_ALG_RSAPSS, ADDR, rsapss, Tss2_MU_TPMS_SIGNATURE_RSA_Marshal,
TPM2_ALG_ECDSA, ADDR, ecdsa, Tss2_MU_TPMS_SIGNATURE_ECC_Marshal,
TPM2_ALG_ECDAA, ADDR, ecdaa, Tss2_MU_TPMS_SIGNATURE_ECC_Marshal,
TPM2_ALG_SM2, ADDR, sm2, Tss2_MU_TPMS_SIGNATURE_ECC_Marshal,
TPM2_ALG_ECSCHNORR, ADDR, ecschnorr, Tss2_MU_TPMS_SIGNATURE_ECC_Marshal,
TPM2_ALG_HMAC, ADDR, hmac, Tss2_MU_TPMT_HA_Marshal,
TPM2_ALG_SPHINCS_SHAKE_256F, ADDR, sphincs,
    Tss2_MU_TPMS_SIGNATURE_SPHINCS_Marshal)

TPMU_UNMARSHAL2(TPMU_SIGNATURE,
TPM2_ALG_RSASSA, rsassa, Tss2_MU_TPMS_SIGNATURE_RSA_Unmarshal,
TPM2_ALG_RSAPSS, rsapss, Tss2_MU_TPMS_SIGNATURE_RSA_Unmarshal,
TPM2_ALG_ECDSA, ecdsa, Tss2_MU_TPMS_SIGNATURE_ECC_Unmarshal,
TPM2_ALG_ECDAA, ecdaa, Tss2_MU_TPMS_SIGNATURE_ECC_Unmarshal,

```

```

TPM2_ALG_SM2, sm2, Tss2_MU_TPMS_SIGNATURE_ECC_Unmarshal,
TPM2_ALG_ECSCNORR, ecschnorr, Tss2_MU_TPMS_SIGNATURE_ECC_Unmarshal,
TPM2_ALG_HMAC, hmac, Tss2_MU_TPMT_HA_Unmarshal,
TPM2_ALG_SPHINCS_SHAKE_256F, sphincs,
    Tss2_MU_TPMS_SIGNATURE_SPHINCS_Unmarshal)

TPMU_MARSHAL2(TPMU_SENSITIVE_COMPOSITE,
TPM2_ALG_RSA, ADDR, rsa, Tss2_MU_TPM2B_PRIVATE_KEY_RSA_Marshal,
TPM2_ALG_ECC, ADDR, ecc, Tss2_MU_TPM2B_ECC_PARAMETER_Marshal,
TPM2_ALG_KEYEDHASH, ADDR, bits, Tss2_MU_TPM2B_SENSITIVE_DATA_Marshal,
TPM2_ALG_SYMCIPHER, ADDR, sym, Tss2_MU_TPM2B_SYM_KEY_Marshal,
TPM2_ALG_SPHINCS_SHAKE_256F, ADDR, sphincs,
    Tss2_MU_TPM2B_PRIVATE_KEY_SPHINCS_Marshal)

TPMU_UNMARSHAL2(TPMU_SENSITIVE_COMPOSITE,
TPM2_ALG_RSA, rsa, Tss2_MU_TPM2B_PRIVATE_KEY_RSA_Unmarshal,
TPM2_ALG_ECC, ecc, Tss2_MU_TPM2B_ECC_PARAMETER_Unmarshal,
TPM2_ALG_KEYEDHASH, bits, Tss2_MU_TPM2B_SENSITIVE_DATA_Unmarshal,
TPM2_ALG_SYMCIPHER, sym, Tss2_MU_TPM2B_SYM_KEY_Unmarshal,
TPM2_ALG_SPHINCS_SHAKE_256F, sphincs,
    Tss2_MU_TPM2B_PRIVATE_KEY_SPHINCS_Unmarshal)

```

This code tells which structures to serialize and deserialize based on the type of algorithm chosen and how to pass them, allowing communication between TPM and the software stack.

The modifications made to `tpm2-tss` are symmetric to those implemented in the TPM, as ensuring this symmetry is crucial. Without this alignment, the TPM would not be able to recognize and correctly utilize the implemented algorithm, compromising interoperability and proper system functionality.

In the case of `tpm2-tools`, several commands were modified to allow the management of post-quantum keys and signatures. Specifically, the `tpm2_createprimary`, `tpm2_create`, `tpm2_quote`, `tpm2_sign`, `tpm2_createek` and `tpm2_createak` commands were updated to support these new algorithms, introducing specific options. The main modified file has been `tpm2_alg_util.c`, in which all the functions are inserted for the various algorithms. In particular, the `handle_object` function has been modified, whose task is to call the correct parameter management function.

```

static alg_parser_rc handle_object(const char *object, TPM2B_PUBLIC
    *public) {

    if (!strcmp(object, "rsa", 3)) {
        object += 3;
        return handle_rsa(object, public);
    } else if (!strcmp(object, "ecc", 3)) {
        object += 3;
        return handle_ecc(object, public);
    } else if (!strcmp(object, "aes", 3)) {
        object += 3;
        return handle_aes(object, public);
    } else if (!strcmp(object, "camellia", 8)) {
        object += 8;
        return handle_camellia(object, public);
    } else if (!strcmp(object, "sm4", 3)) {
        object += (object[3] == '_') ? 4 : 3;
        return handle_sm4(object, public);
    } else if (!strcmp(object, "hmac")) {
        return handle_hmac(public);
    } else if (!strcmp(object, "xor")) {
        return handle_xor(public);
    } else if (!strcmp(object, "keyedhash")) {

```

```
        return handle_keyedhash(public);
    } else if (!strcmp(object, "sphincs")){
        return handle_sphincs(public);
    }

    return alg_parser_rc_error;
}
```

The *handle_sphincs* function has been added, which must set the parameters necessary to carry out the algorithm.

```
static alg_parser_rc handle_sphincs(TPM2B_PUBLIC *public) {
    public->publicArea.type = TPM2_ALG_SPHINCS_SHAKE_256F;
    TPMS_RSA_PARMS *r = &public->publicArea.parameters.rsaDetail;
    r->exponent = 0;
    r->keyBits = 2048;
    r->scheme.scheme = TPM2_ALG_RSAES;

    return alg_parser_rc_continue;
}
```

In this case, it was necessary to explicitly specify the variable type *TPMS_RSA_PARMS*, as the system would otherwise be unable to correctly handle the parameters for the *TPM2_ALG_SPHINCS_SHAKE_256F* algorithm. Future implementations may introduce new parameter types to be used in similar contexts. Additionally, the required handling was added within the *tpm2_alg_util_for_each_alg* function.

```
{ .name = "sphincs", .id= TPM2_ALG_SPHINCS_SHAKE_256F, .flags =
    tpm2_alg_util_flags_asymmetric|tpm2_alg_util_flags_base },
```

Thanks to this line of code, it has been possible to make the new string recognized by all the commands of tpm2-tools and use the new command line commands. The modifications in tpm2-tools also included the propagation of the newly introduced constant, as it affects a switch case responsible for setting algorithm parameters. While in this implementation the addition was only necessary for a single post-quantum algorithm, this change lays the groundwork for future implementations where multiple algorithms with different parameters could be supported.

Bibliography

- [1] P. Shor, “Algorithms for quantum computation: discrete logarithms and factoring”, Proceedings 35th Annual Symposium on Foundations of Computer Science, 1994, pp. 124–134, DOI [10.1109/SFCS.1994.365700](https://doi.org/10.1109/SFCS.1994.365700)
- [2] S. Paul, F. Schick, and J. Seedorf, “TPM-Based Post-Quantum Cryptography: A Case Study on Quantum-Resistant and Mutually Authenticated TLS for IoT Environments”, Proceedings of the 16th International Conference on Availability, Reliability and Security, New York, NY, USA, 2021, DOI [10.1145/3465481.3465747](https://doi.org/10.1145/3465481.3465747)
- [3] L. Fiolhais, P. Martins, and L. Sousa, “Software Emulation of Quantum Resistant Trusted Platform Modules”, August 2020, DOI [10.5281/zenodo.3979200](https://doi.org/10.5281/zenodo.3979200)
- [4] L. Fiolhais and L. Sousa, “QR TPM in Programmable Low-Power Devices”, arXiv preprint arXiv:2309.17414, September 2023
- [5] NIST, “NIST to Standardize Encryption Algorithms That Can Resist Attack by Quantum Computers”, <https://www.nist.gov/news-events/news/2023/08/nist-standardize-encryption-algorithms-can-resist-attack-quantum-computers>
- [6] N. Asokan, J.-E. Ekberg, K. Kostianen, A. Rajan, C. Rozas, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, “Mobile Trusted Computing”, Proceedings of the IEEE, vol. 102, no. 8, 2014, pp. 1189–1206, DOI [10.1109/JPROC.2014.2332007](https://doi.org/10.1109/JPROC.2014.2332007)
- [7] GlobalPlatform, “GlobalPlatform Card Specification v2.3.1”. GlobalPlatform, March 2018. Document Number: GPC_SPE_034. Available at <https://globalplatform.org/specs-library/card-specification-v2-3-1/>
- [8] M. Sabt, M. Achemlal, and A. Bouabdallah, “Trusted Execution Environment: What It is, and What It is Not”, 2015 IEEE Trustcom/BigDataSE/ISPA, 2015, pp. 57–64, DOI [10.1109/Trustcom.2015.357](https://doi.org/10.1109/Trustcom.2015.357)
- [9] A. Khalid, S. McCarthy, M. O’Neill, and W. Liu, “Lattice-based Cryptography for IoT in A Quantum World: Are We Ready?”, 2019 IEEE 8th International Workshop on Advances in Sensors and Interfaces (IWASI), 2019, pp. 194–199, DOI [10.1109/IWASI.2019.8791343](https://doi.org/10.1109/IWASI.2019.8791343)
- [10] Q. Zhu, Q. Chen, Y. Liu, Z. Akhtar, and K. Siddique, “Investigating TrustZone: A Comprehensive Analysis”, Security and Communication Networks, vol. 2023, no. 1, 2023, p. 7369634, DOI <https://doi.org/10.1155/2023/7369634>
- [11] S. Pinto and N. Santos, “Demystifying Arm TrustZone: A Comprehensive Survey”, ACM Comput. Surv., vol. 51, jan 2019, DOI [10.1145/3291047](https://doi.org/10.1145/3291047)
- [12] V. Mavroeidis, K. Vishi, M. D., and A. Jøsang, “The Impact of Quantum Computing on Present Cryptography”, International Journal of Advanced Computer Science and Applications, vol. 9, no. 3, 2018, DOI [10.14569/ijacsa.2018.090354](https://doi.org/10.14569/ijacsa.2018.090354)
- [13] A. Menezes, P. C. van Oorschot, and S. A. Vanstone, “Handbook of Applied Cryptography”, CRC Press, 1996, ISBN: 0-8493-8523-7
- [14] R. P. Feynman, “Simulating Physics with Computers”, International Journal of Theoretical Physics, vol. 21, no. 6–7, 1982, pp. 467–488, DOI [10.1007/BF02650179](https://doi.org/10.1007/BF02650179)
- [15] T. Monz, D. Nigg, E. A. Martinez, M. F. Brandl, P. Schindler, R. Rines, S. X. Wang, I. L. Chuang, and R. Blatt, “Realization of a scalable Shor algorithm”, Science, vol. 351, March 2016, pp. 1068–1070, DOI [10.1126/science.aad9480](https://doi.org/10.1126/science.aad9480)
- [16] C. Cheng, R. Lu, A. Petzoldt, and T. Takagi, “Securing the Internet of Things in a Quantum World”, IEEE Communications Magazine, vol. 55, 02 2017, pp. 116–120, DOI [10.1109/MCOM.2017.1600522CM](https://doi.org/10.1109/MCOM.2017.1600522CM)

- [17] L. K. Grover, “A Fast Quantum Mechanical Algorithm for Database Search”, Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC), 1996, pp. 212–219, DOI [10.1145/237814.237866](https://doi.org/10.1145/237814.237866)
- [18] M. A. López and M. M. da Silva, “Quantum Technologies: Digital Transformation, Social Impact, and Cross-sector Disruption”, 2019, DOI [http://dx.doi.org/10.18235/0001613](https://dx.doi.org/10.18235/0001613)
- [19] N. Liu and P. Rebentrost, “Quantum machine learning for quantum anomaly detection”, Physical Review A, vol. 97, April 2018, DOI [10.1103/physreva.97.042315](https://doi.org/10.1103/physreva.97.042315)
- [20] A. Alomari and S. A. Kumar, “Securing IoT systems in a post-quantum environment: Vulnerabilities, attacks, and possible solutions”, Internet of Things, vol. 25, 2024, p. 101132, DOI <https://doi.org/10.1016/j.iot.2024.101132>
- [21] G. Brassard, P. Høyer, and A. Tapp, “Quantum cryptanalysis of hash and claw-free functions”, SIGACT News, vol. 28, June 1997, pp. 14–19, DOI [10.1145/261342.261346](https://doi.org/10.1145/261342.261346)
- [22] D. Simon, “On the power of quantum computation”, Proceedings 35th Annual Symposium on Foundations of Computer Science, 1994, pp. 116–123, DOI [10.1109/SFCS.1994.365701](https://doi.org/10.1109/SFCS.1994.365701)
- [23] L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang, “Experimental realization of Shor’s quantum factoring algorithm using nuclear magnetic resonance”, Nature, vol. 414, December 2001, pp. 883–887, DOI [10.1038/414883a](https://doi.org/10.1038/414883a)
- [24] K. Bertels, A. Sarkar, T. Hubregtsen, M. Serrao, A. A. Mouedenne, A. Yadav, A. Krol, I. Ashraf, and C. G. Almudever, “Quantum Computer Architecture Toward Full-Stack Quantum Accelerators”, IEEE Transactions on Quantum Engineering, vol. 1, no. 1, 2020, pp. 1–12, DOI [10.1109/TQE.2020.2988277](https://doi.org/10.1109/TQE.2020.2988277)
- [25] M. Mosca, “Cybersecurity in an Era with Quantum Computers: Will We Be Ready?”, IEEE Security & Privacy, vol. 16, no. 5, 2018, pp. 38–41, DOI [10.1109/MSP.2018.3761723](https://doi.org/10.1109/MSP.2018.3761723)
- [26] NIST, “What Is Post-Quantum Cryptography?”, <https://www.nist.gov/cybersecurity/what-post-quantum-cryptography>
- [27] E. Malygina, A. Kutsenko, S. Novoselov, N. Kolesnikov, A. Bakharev, I. Khilchuk, A. Shaporenko, and N. Tokareva, “Post-Quantum Cryptosystems: Open Problems and Solutions. Lattice-Based Cryptosystems”, Journal of Applied and Industrial Mathematics, vol. 17, 02 2024, pp. 767–790, DOI [10.1134/S1990478923040087](https://doi.org/10.1134/S1990478923040087)
- [28] D. Micciancio and O. Regev, “Lattice-based Cryptography”, pp. 147–191. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009
- [29] “PQ-CRYSTALS”, <https://pq-crystals.org/>, Accessed on 21 January 2025
- [30] T. Pornin, “New Efficient Constant-Time Implementations of Falcon”, <https://falcon-sign.info>, Accessed on 21 January 2025
- [31] C. Q. Choi, ““Quantum-Safe” Crypto Hacked by 10-Year-Old PC”, <https://spectrum.ieee.org/quantum-safe-encryption-hacked>
- [32] C. Shen, H. Zhang, H. Wang, J. Wang, B. Zhao, F. Yan, F. Yu, L. Zhang, and M. Xu, “Research on trusted computing and its development”, SCIENCE CHINA Information Sciences, vol. 53, 03 2010, pp. 405–433, DOI [10.1007/s11432-010-0069-x](https://doi.org/10.1007/s11432-010-0069-x)
- [33] C. Mundie, P. de Vries, P. Haynes, and M. Corwine, “Trustworthy computing”, tech. rep., Technical report, 10, 2002
- [34] U.S. Department of Defense, “Department of Defense Trusted Computer System Evaluation Criteria (TCSEC)”, Tech. Rep. DoD 5200.28-STD, National Computer Security Center (NCSC), December 1985. Part of the Orange Book series. Available at <https://fas.org/irp/nsa/rainbow/std001.htm>
- [35] Trusted Computing Group, “Trusted Computing Group (TCG)”, 2024, Available at <https://trustedcomputinggroup.org/>
- [36] Trusted Computing Group, “Trusted Computing Group Specifications”, 2024. Available at <https://trustedcomputinggroup.org/specs/>
- [37] A.-R. Sadeghi, C. Stübke, and N. Pohlmann, “European Multilateral Secure Computing Base Open Trusted Computing for You and Me”, 2004
- [38] ISO/IEC, “Common Criteria for Information Technology Security Evaluation”, 2017. Version 3.1 Revision 5
- [39] Y. Kim and E. Kim, “hTPM: Hybrid Implementation of Trusted Platform Module”, Proceedings of the 1st ACM Workshop on Workshop on Cyber-Security Arms Race, New York, NY, USA, 2019, pp. 3–10, DOI [10.1145/3338511.3357348](https://doi.org/10.1145/3338511.3357348)

- [40] K. Kursawe, D. Schellekens, and B. Preneel, “Analyzing Trusted Platform Communication”, ECRYPT Workshop, CRASH – CRYPTOGRAPHIC Advances in Secure Hardware, 2005
- [41] Trusted Computing Group, “TCG TPM Specification Version 1.2”, November 2003. Available at <https://trustedcomputinggroup.org/resource/tpm-main-specification/>
- [42] Trusted Computing Group, “TCG TPM 2.0 Library Specification”, March 2016. Available at <https://trustedcomputinggroup.org/resource/tpm-library-specification/>
- [43] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten, “fTPM: a software-only implementation of a TPM chip”, Proceedings of the 25th USENIX Conference on Security Symposium, USA, 2016, pp. 841–856
- [44] Global Platform, “TEE System Architecture v1.3”, https://globalplatform.org/wp-content/uploads/2022/05/GPD_SPE_009-GPD_TEE_SystemArchitecture_v1.3_PublicRelease_signed.pdf
- [45] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. McCune, “Trustworthy Execution on Mobile Devices: What Security Properties Can My Mobile Platform Give Me?”, Trust and Trustworthy Computing (A. P. Fuchs and P. L. P. da Silva, eds.), pp. 159–178, Springer, Berlin, 2012, DOI [10.1007/978-3-642-30921-2_10](https://doi.org/10.1007/978-3-642-30921-2_10)
- [46] K. Kostiainen, N. Asokan, and J.-E. Ekberg, “Practical Property-Based Attestation on Mobile Devices”, Trust and Trustworthy Computing, 2011, pp. 78–92, DOI [10.1007/978-3-642-21599-5_6](https://doi.org/10.1007/978-3-642-21599-5_6)
- [47] J. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, “SeCReT: Secure Channel Between Rich Execution Environment and Trusted Execution Environment”, Proceedings of the Network and Distributed System Security Symposium (NDSS), 2015, DOI [10.14722/ndss.2015.23189](https://doi.org/10.14722/ndss.2015.23189)
- [48] J. Jang and B. B. Kang, “Retrofitting the Partially Privileged Mode for TEE Communication Channel Protection”, IEEE Transactions on Dependable and Secure Computing, 2018, pp. 1–1, DOI [10.1109/TDSC.2018.2840709](https://doi.org/10.1109/TDSC.2018.2840709)
- [49] J. González and P. Bonnet, “Versatile Endpoint Storage Security with Trusted Integrity Modules”, technical report, IT University, 2014
- [50] J. Winter, “Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms”, Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing, New York, NY, USA, 2008, DOI [10.1145/1456455.1456460](https://doi.org/10.1145/1456455.1456460)
- [51] P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves, “Implementing Embedded Security on Dual-Virtual-CPU Systems”, IEEE Design & Test of Computers, vol. 24, November 2007, pp. 582–591, DOI [10.1109/MDT.2007.178](https://doi.org/10.1109/MDT.2007.178)
- [52] N. L. P. Jr., T. Fraser, J. Molina, and W. A. Arbaugh, “Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor”, Proceedings of the 13th USENIX Security Symposium, San Diego, CA, 2004
- [53] M. S. Kirkpatrick, G. Ghinita, and E. Bertino, “Resilient Authenticated Execution of Critical Applications in Untrusted Environments”, IEEE Transactions on Dependable and Secure Computing, vol. 9, no. 4, 2012, pp. 597–610, DOI [10.1109/TDSC.2012.51](https://doi.org/10.1109/TDSC.2012.51)
- [54] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, “Flicker: An Execution Infrastructure for TCB Minimization”, Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, New York, NY, USA, 2008, pp. 315–328, DOI [10.1145/1352592.1352625](https://doi.org/10.1145/1352592.1352625)
- [55] T. Alves and D. Felton, “TrustZone: Integrated Hardware and Software Security - Enabling Trusted Computing in Embedded Systems”, July 2004, Available online at: http://www.arm.com/pdfs/TZ_Whitepaper.pdf
- [56] ARM Ltd., “TrustZone Technology Overview”, Introduction available at: http://www.arm.com/products/esd/trustzone_home.html
- [57] M. Kuhne, S. Sridhara, A. Bertschi, N. Dutly, S. Capkun, and S. Shinde, “Aster: Fixing the Android TEE Ecosystem with Arm CCA”, arXiv preprint arXiv:2407.16694, 2024, DOI [10.48550/arXiv.2407.16694](https://doi.org/10.48550/arXiv.2407.16694)
- [58] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee, “SeCloak: ARM Trustzone-based Mobile Peripheral Control”, New York, NY, USA, 2018, pp. 1–13, DOI [10.1145/3210240.3210334](https://doi.org/10.1145/3210240.3210334)
- [59] ARM Ltd., “ARM CoreLink TZC-400 TrustZone Address Space Controller”, 2014. Available online at: <https://developer.arm.com/documentation>

- [60] ARM Ltd., “Cortex-A9 Technical Reference Manual - 6.3 Memory Access Sequence”. Accessed 12 September 2018
- [61] Kingston Technology, “Kingston Embedded Solutions”, <https://www.kingston.com/en/embedded/emmc>, Accessed on January 13, 2025
- [62] A. K. Reddy, P. Paramasivam, and P. B. Vemula, “Mobile Secure Data Protection Using eMMC RPMB Partition”, 2015 International Conference on Computing and Network Communications (CoCoNet), 2015, pp. 946–950, DOI [10.1109/CoCoNet.2015.7411305](https://doi.org/10.1109/CoCoNet.2015.7411305)
- [63] X. Zhu *et al.*, “Analysis and Research on TrustZone Technology”, Journal of Computer Security, vol. 30, no. 2, 2023, pp. 123–145, DOI [10.1155/2023/7369634](https://doi.org/10.1155/2023/7369634)
- [64] ARM Limited, “SMC Calling Convention System Software on ARM®Platforms”, 2016
- [65] P. Greenhalgh, “big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7”, ARM White Paper 17, ARM Ltd., 2011. Available online at: <https://www.arm.com/whitepapers>
- [66] ARM Ltd., “ARM1176JZF-S Technical Reference Manual - 2.12.13 Secure Monitor Call (SMC)”, n.d. Accessed on January 13, 2025
- [67] X. Li, Z. Li, Z.-h. Li, and Z.-j. Li, “Extension Implementation of TCM in the Embedded System Based on FPGA”, Proceedings of the 2013 International Conference on Computer Sciences and Applications (CSA), 2013, pp. 180–183, DOI [10.1109/CSA.2013.180](https://doi.org/10.1109/CSA.2013.180)
- [68] ARM Ltd., “ARM1176JZF-S Technical Reference Manual, Revision: r0p7”, 2008. Available online at: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301g/DDI0301G_arm1176jzfs_r0p7_trm.pdf
- [69] Y. Chen, Y. Zhang, Z. Wang, and T. Wei, “Downgrade Attack on TrustZone”, arXiv preprint arXiv:1707.05082, 2017, DOI [10.48550/arXiv.1707.05082](https://doi.org/10.48550/arXiv.1707.05082)
- [70] A. K. Sarker, M. K. Islam, and Y. Tian, “MVAM: Multi-variant Attacks on Memory for IoT Trust Computing”, Proceedings of the 2023 Workshop on CPS & IoT Security and Privacy, 2023, pp. 13–18, DOI [10.1145/3576914.3587486](https://doi.org/10.1145/3576914.3587486)
- [71] OP-TEE, “OP-TEE OS”, https://github.com/OP-TEE/optee_os, n.d., Accessed: January 13, 2025
- [72] OP-TEE, “OP-TEE Supplicant”, https://github.com/OP-TEE/optee_client/tree/master/tee-supplciant, Accessed: January 13, 2025
- [73] OP-TEE, “OP-TEE Build”, <https://github.com/OP-TEE/build>, n.d., Accessed: January 13, 2025
- [74] OP-TEE, “OP-TEE Sanity Test Suite”, <https://github.com/OP-TEE/sanity-testing>, n.d., Accessed: January 13, 2025
- [75] QEMU Project, “QEMU”, <https://www.qemu.org>, Accessed: January 13, 2025
- [76] D. Stebila and M. Mosca, “Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project”, Selected Areas in Cryptography - SAC 2016, 2017, pp. 14–37, DOI [10.1007/978-3-319-69453-5_2](https://doi.org/10.1007/978-3-319-69453-5_2)
- [77] “Open Source Liboqs Library”, <https://openquantumsafe.org/liboqs>, Accessed on 21 January 2025
- [78] OP-TEE Project, “OP-TEE Documentation”, 2024, Accessed: 2024-03-09
- [79] tpm2-software project, “tpm2-tools Documentation”, <https://tpm2-tools.readthedocs.io/>, 2024, Accessed: March 22, 2025
- [80] Open Quantum Safe Project, “Liboqs: C library for quantum-safe cryptography”, <https://github.com/open-quantum-safe/liboqs>, Accessed: 2025-01-25