POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Quantum-safe Remote Attestation

Supervisor
Prof. Antonio Lioy
Ing. Grazia D'Onghia

Candidate
Gabriele DERAJ

April 2025

To my family

Summary

In recent years, quantum computing has been rapidly developing and its impact on cybersecurity is a major concern. Problems that were once considered impossible to solve using traditional computational platforms have now become manageable for quantum computers. All encryption methods based on RSA and other classical algorithms are at serious risk: RSA and elliptic curve cryptography rely on mathematical problems such as integer factorization and discrete logarithms, and quantum algorithms like Shor's and Grover's can crack them in a really fast way. This work focuses on the impact of quantum computing on the field of remote attestation. Remote attestation is a process used by an external entity to verify the integrity and trustworthiness of a computational node. It ensures that a system has not been tampered with, and plays a key role in protecting sensitive information. However, remote attestation mechanisms rely on cryptographic primitives that are vulnerable to quantum attacks. Soon, adversaries equipped with quantum computers could forge cryptographic proofs, compromise key exchange mechanisms, and fake the very foundation of trust in remote attestation systems. This makes the transition to quantum-resistant cryptographic algorithms a necessity to maintain the security of trusted computing environments. The proposed solution implements the integration of post-quantum cryptographic algorithms into the Keylime framework. Keylime is an open source platform that provides a modular environment for remote attestation, using the Trusted Platform Module (TPM) for cryptographic tasks. To ensure that the system remains secure against quantum threats, this thesis proposes the SPHINCS+ signature scheme, a quantum-safe algorithm that is among the most promising candidates in the post-quantum cryptography standardization process. The tests also discuss the challenges of the transition to post-quantum cryptography, such as the larger signature sizes of these algorithms and the computational load caused by these more complex operations. Despite these challenges, the work provides valuable insight into the use of quantum-safe techniques in a remote attestation environment.

Acknowledgements

Thanks.

Contents

Chapter 1

Introduction

1.1 The impact of Quantum computing

Until recently, the Information and Communication Technology (ICT) industry considered transactions involving information exchange across electronic networks secure when encrypted with conventional cryptographic systems. However, recent advances in quantum computing research have significantly threatened this assumption.
Problems considered difficult or impossible to solve using traditional computational platforms become manageable for quantum computers. Consequently, any encrypted information is at risk of eavesdropping and attacks by future adversaries equipped with quantum computing capabilities. This implies that encrypted data stored in databases, even for 25 years, could eventually be exposed to those with quantum computing access. This phenomenon is known as "Store Now, Decrypt Later" (SNDL) [1]. SNDL is a cryptographic strategy that involves long-term encrypted data storage to decrypt once quantum-resistant algorithms are developed.
This risk covers sensitive information such as bank account numbers, personal identity details, military security data, and other confidential information. Without quantum-safe encryption, all data that have been or will be transmitted over a network remain vulnerable to eavesdropping and potential public disclosure[2].

Quantum computing theory was first introduced as a concept in 1982 by Richard Feynman [3] and is considered the destructor of the present modern asymmetric cryptography. In addition, specific quantum algorithms can also affect symmetric cryptography 1.3.2. It appears that even elliptic curve cryptography, which is considered presently the most secure and efficient scheme, is weak against quantum computers. Consequently, it is necessary to adopt cryptographic algorithms resistant to quantum computations. A Quantum computer is no longer a hypothetical idea. It is defined as a Cryptographically Relevant Quantum Computer (CRQC) and is considered one of the world's most important technologies.
There is a race among countries to obtain supremacy in quantum technology with a quantum computer of a sufficient number of qubits and fault tolerance. In a nutshell, quantum computers threaten the main goal of every secure and authenticated communication because they can do computations that classical (conventional) computers cannot. Consequently, quantum computers can break cryptographic keys quickly by calculating or exhaustively searching all secret keys, allowing an eavesdropper to intercept the communication channel between authentic parties [4].

1.2 Quantum computing vs classical computing

The laws of physics observable and comprehensible through everyday experiences are known as classical physics. However, all phenomena described by classical physics at the macroscopic level can also be described by quantum physics at the nanoscale, governed by the principles of quantum mechanics [5]. In recent decades, researchers have discovered that the unique behaviors allowed by quantum mechanics at the microscopic scale can be harnessed to create computers from new

materials. These quantum computers feature hardware that significantly differs in form and function from the classical computers commonly used in homes and offices today. Operating under the principles of quantum mechanics, quantum computers can perform calculations in ways that go beyond the capabilities of conventional classical computing, presenting new paradigms for processing information.

1.2.1 Qubits and their properties

In a traditional computer, the fundamental blocks are called bits and can be observed only in two states: 0 and 1. Quantum computers instead use quantum bits, also generally referred to as qubits [6]. Qubits are particles that can exist not only in the 0 and 1 state but in both simultaneously, known as superposition $\langle 0|1\rangle$, mathematically represented as

$$\alpha\,|0\rangle + \beta\,|1\rangle$$

where α and β are complex coefficients. A particle collapses into one of these states when inspected. Quantum computers take advantage of this to solve complex problems. An operation on a qubit in superposition acts on both values at the same time. Another physical phenomenon used in quantum computing is quantum entanglement. When two qubits are entangled, their quantum state can no longer be described independently of each other but as a single object with four different states. In addition, if one of the two qubits changes, the entangled qubit will change too, regardless of the distance between them. This leads to true parallel processing power. When the number of entangled qubits increases, the number of values that can be processed in a single operation grows exponentially. This means that an n-qubit quantum computer can perform 2^n operations at the same time. We can say that quantum computing is based on the following three features of quantum states:

- **Superposition**: quantum systems can exist in two states simultaneously. When measurement is performed, the qubit collapses into one of the possible definite states. This means that the qubit, which was in a superposition of states before measurement, now assumes a concrete value (0 or 1). The probability of obtaining a particular measurement result is determined by the coefficients α and β of the superposition. This phenomenon is known as "collapse of the wave function"[7].
- **Entanglement**: as previously said, it is a phenomenon where the state of particles can be described concerning each other. Measurement performed on one entangled particle will immediately influence the other, irrespective of the distance between them.
- **Interference**: the fundamental idea in quantum computing is to control the probability of qubits collapsing into a particular measurement state. Quantum interference, which comes from superposition, allows controlling the measurement of a qubit toward a desired state or group of states.

Quantum computers can show their superiority over classical computers only when using algorithms that leverage quantum parallelism. For example, a quantum computer would not be any faster than a traditional computer in multiplication. It is important to notice that quantum computers are still at an experimental stage and are not yet widely available. There are many challenges in quantum computing that many researchers are working on:

- quantum algorithms are mainly probabilistic: in one operation, a quantum computer returns many solutions where only one is correct.
- qubits are importantly susceptible to noise and electromagnetic couplings, which can lead to measurement errors.
- qubits can retain their quantum state for a short period. Researchers at the University of New South Wales in Australia [4] have created two different types of qubits (Phosphorous atom and an Artificial atom). Phosphorous atom has 99.99% accuracy (1 error every 10,000

quantum operations). Their qubits can remain in superposition for a total of 35 seconds. Moreover, to maintain long coherence, qubits need not only to be isolated from the external environment but also to be kept at temperatures close to absolute zero, as higher temperatures contribute to increased noise.

1.2.2 The EPR paradox and the non-locality principle

The EPR (Einstein-Podolsky-Rosen) paradox is an experiment proposed in 1935 by Einstein, Podolsky, and Rosen to highlight contradictions between quantum mechanics and local realism. It is based on two particles in an entangled quantum state, where the state of each depends instantaneously on the other, regardless of distance. Einstein and colleagues considered entanglement incompatible with special relativity and the principle of locality, arguing that quantum mechanics was incomplete and that hidden local variables must exist. However, in 1964, John Bell introduced inequalities that, if experimentally violated, would disprove the existence of such variables. In 2022, Alain Aspect, John F. Clauser, and Anton Zeilinger were awarded the Nobel Prize for experimentally violating Bell's inequalities, proving that quantum mechanics is a complete theory and that the non-locality of some of its phenomena is real.

1.3 Major threats to actual cryptosystems

In the world of keeping information safe, quantum computing brings both big chances and big problems. Two special computer methods, known as Shor's algorithm [8] and Grover's algorithm [9], are leading the way in this new era. In this part, we're talking about Shor's algorithm and Grover's algorithm in simple terms, looking at how they work and what they mean for keeping secrets safe and solving problems quickly. Learning about these unique methods helps us understand how quantum computing can transform the way we protect information and perform complex calculations.

1.3.1 Shor's algorithm

In the popular RSA public-key system, the public key is a product $N = pq$ of two secret prime numbers p and q. The security of RSA relies critically on the difficulty in finding the factors p and q of N. Peter Shor introduced a fast quantum algorithm to find the prime factorization of any positive integer N. In 1994, according to [4], Shor proved that factoring large integers would change fundamentally with a quantum computer. Shor's algorithm can make modern asymmetric cryptography collapse since it is based on large prime integer factorization or the discrete logarithm problem. How does it work? Suppose we want to find the prime factors of 15. To do so, we need a 4-qubit register. Number 15 in binary is 1111, so a 4-qubit register is enough to calculate the prime factorization of this number. The algorithm does the following :

- $n = 15$, is the number we want to factorize
- x is a random number such as $1 < x < n - 1$
- x is raised to the power contained in the register (every possible state) and then divided by n. The remainder from this operation is stored in a second 4-qubit register. The second register now contains the superposition results. Let's assume that $x = 2$ is larger than 1 and smaller than 14.

If we raise x to the powers of the 4-qubit register, which is a maximum of 15, and divide by 15, what we observe in the results is a repeating sequence of 4 numbers (1,2,4,8). We can confidently say then that $f = 4$, which is the sequence when $x = 2$ and $n = 15$. The value f can be used to calculate a possible factor with the following equation: Possible factor: $P = x^{(f/2)} - 1$. If we get a result that is not a prime number, we repeat the calculation with different f values. All public key algorithms used today are based on two mathematical problems, the aforementioned factorization

of large numbers (e.g., RSA) and the calculation of discrete logarithms (e.g., DSA signatures and ElGamal encryption). Both have similar mathematical structures and can be broken with Shor's algorithm easily. Recent algorithms based on elliptic curves (such as ECDSA) use a modification of the discrete logarithm problem that makes them equally weak against quantum computers.

1.3.2 Grover's algorithm

Grover's algorithm uses quantum computers to search unsorted databases. The algorithm can find a specific entry in an unsorted database of N entries in $\sqrt{N}$ searches. In comparison, a conventional computer would need $N/2$ searches to find the same entry. Let's consider the impact of a possible application of Grover's algorithm to crack Data Encryption Standard (DES), which relies on a 56-bit key. The algorithm needs only 185 searches to find the key. Currently, to prevent password cracking we increase the number of key bits (larger key space); as a result, the number of searches needed to crack a password increases exponentially. The National Institute of Standards and Technology (NIST) points out that if the key sizes are sufficient, symmetric cryptographic schemes (specifically the Advanced Encryption Standard-AES) are resistant to quantum computers. For this reason, quantum computing is considered a minor threat to symmetric cryptography.

It is better to describe Grover's algorithm as searching for roots of a function f: searching for solutions x to the equation $f(x) = 0$. Grover's speedup from N to $\sqrt{N}$ is not as devastating as Shor's speedup. Furthermore, each of Grover's N quantum evaluations must wait for the previous one to finish. On the other hand, if qubit operations are small enough and fast enough, then Grover's algorithm will threaten many cryptographic systems that aim for 2^{128} security, such as 128-bit AES keys. Therefore, it is recommended to switch to 256-bit AES keys: the extra costs are rarely noticeable. "Information-theoretic" MACs such as GMAC already protect against quantum computers without any modifications: their security analysis already assumes an attacker with unlimited computing power.

1.4 Towards a Quantum-Safe architecture for a Remote Attestation environment

The disruptive potential of quantum computing also extends to the field of cybersecurity, where many widely used cryptographic algorithms are built on mathematical problems that quantum computers could efficiently solve. As detailed in the previous sections, Shor's algorithm compromises cryptographic schemes such as RSA and Elliptic Curve Cryptography (ECC). This growing threat is pushing both researchers and companies to take a fresh look at how secure today's systems are, especially those meant to guarantee the safety and reliability of computing environments. Among these, remote attestation could be a critical security mechanism fundamentally compromised in a post-quantum world.

Remote attestation is a security mechanism that allows an external trusted entity (the verifier) to assess whether a computational node (the attester) is operating in a secure and uncompromised state. This is particularly vital in distributed environments, cloud infrastructures, and edge computing scenarios, where trust cannot be inherently assumed and must be established through verifiable proof of integrity. To ensure the security and reliability of this process, modern systems often rely on specialized hardware components such as Trusted Platform Modules (TPMs). The TPM provides functionalities such as cryptographic key generation, secure storage, and remote verification of system states. One of its main responsibilities is generating proof of system integrity. We have referred to this as "proof", but in the context of remote attestation, this is more precisely called a "quote". The quote is a digitally signed statement that encapsulates the integrity measurements of a system, allowing a remote verifier to assess its trustworthiness. Within a TPM there are some special registers called platform configuration registers (PCR), which store records of the system's configuration over time. When performing a Remote Attestation, the TPM signs the contents of these registers with a cryptographic key, proving that the system is in a known and trusted state and ensuring that the evidence originates from a genuine

and untampered TPM. However, the security of this entire mechanism hinges on the strength of the cryptographic algorithms employed by the TPM.

Currently, TPMs rely on RSA or ECC keys, which are secure against classical attacks but vulnerable to quantum ones. A powerful quantum computer running Shor's algorithm could break these cryptographic methods, allowing attackers to forge valid attestations and compromise the integrity of the remote attestation process. As we said, the TPM is a hardware component, and its cryptographic keys are injected during manufacturing. However, TPMs with hardware support for post-quantum algorithms have not yet been produced, meaning that all existing TPMs are still based on classical cryptography. Because of this, the first step toward a quantum-safe transition must rely on software-based solutions to extend current remote attestation. This approach allows systems to begin adapting to post-quantum security requirements while waiting for future TPMs with built-in support for quantum-resistant algorithms.

The set of cryptographic algorithms that aim to protect systems from the potential threats posed by quantum computers is called post-quantum cryptography. This work explores the integration of post-quantum cryptographic algorithms in the context of remote attestation, using the Keylime framework as the foundation for the implementation. Keylime is an open-source framework designed for trusted computing that facilitates this process by providing a modular and scalable environment for remote attestation. Its design ensures compatibility with modern trusted computing hardware, such as the Trusted Platform Module (TPM), and enables secure attestation processes even in distributed systems.

Keylime operates as a collection of four primary components: the agent, the verifier, the registrar, and the tenant. Each plays a distinct role in the attestation process.

- Agent: the agent is responsible for collecting integrity measurements and generating a TPM-based quote that provides cryptographic evidence of the system's state.
- Verifier: the verifier receives the quote from the agent and checks its validity by comparing it to the expected values. If the integrity measurements align, the verifier confirms the agent's trustworthiness.
- Registrar: the registrar acts as a central database for managing TPM credentials and provides the verifier with the necessary information about registered agents.
- Tenant: the tenant represents the end-user or application that relies on the verifier to assess the agent's trustworthiness before deploying sensitive workloads.

Communication between the agent and the verifier is secured using a TLS channel to prevent eavesdropping or tampering. The TLS implementation can also benefit from quantum-safe key exchange algorithms for added protection.

To make this environment quantum-safe, two main solutions can be performed:

1. **Extending the TPM driver** to support key signing and management based on Post-Quantum Cryptography algorithms. The TPM (Trusted Platform Module) is responsible for generating and signing quotes during the attestation process. Nowadays, no TPM includes hardware implementation of Post-Quantum Cryptography algorithms. Current TPMs are designed to work with classical cryptographic standards such as RSA and ECC, which are potentially vulnerable to future quantum computing attacks. To withstand future attacks, the TPM driver must be updated to support cryptographic algorithms resistant to quantum attacks, for example, enabling the generation of post-quantum keypairs and the usage of post-quantum algorithms, and to have a post-quantum signature, at least at the kernel level.
2. **Extending the attestation agent and trust manager** to integrate and handle Post-Quantum Cryptography. This second extension is the core of this work and will focus on updating the attestation flow to manage quotes signed with PQ algorithms. The attestation agent, running on the attester machine, must be able to communicate with the extended TPM driver to gather and send signed quotes to the trust manager. The trust manager, in turn, must verify the integrity of the received PQ signatures. These enhancements will

provide a software-based mechanism to ensure that platforms can start transitioning towards quantum-safe cryptographic operations, even before dedicated hardware becomes available. This is a solution implemented at the user-space level.

While the ideal long-term solution would be integrating post-quantum cryptography directly into the TPM, making the attestation process inherently quantum-safe at the hardware level, this is not yet possible due to the current limitations of TPM technology. In the meantime, the most practical approach is to extend the attestation agent and perform post-quantum signing at the software level. Although this could introduce additional computational overhead and potential attack surfaces, it provides an immediate layer of quantum resistance without requiring changes to existing TPM hardware. This second extension is the core of this work and will focus on

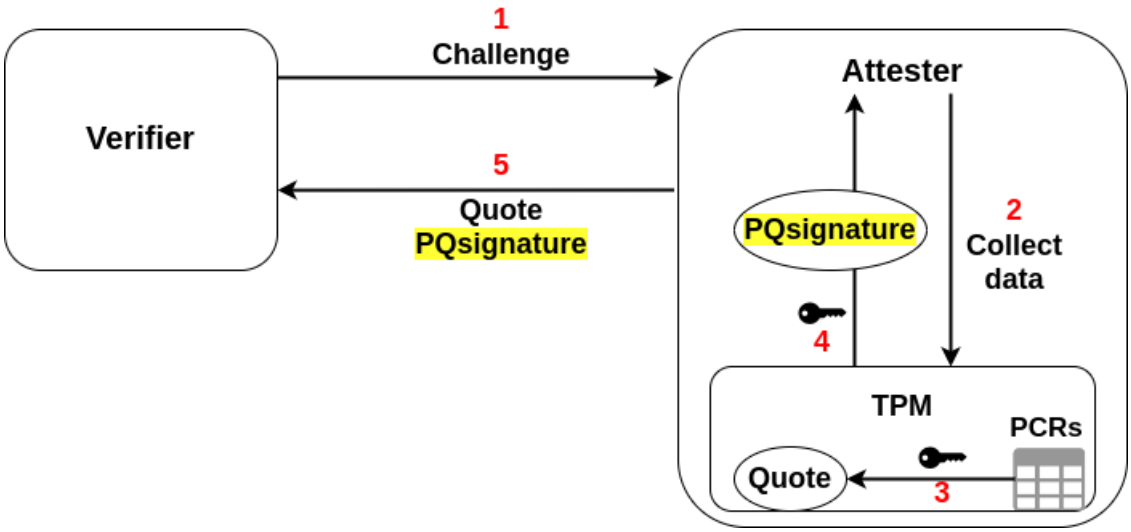

Figure 1.1. PQ-enhanced remote attestation worflow

updating the attestation flow to include post-quantum signatures. The attestation process is thus modified as shown in 1.1: the verifier requests the attester to prove the integrity. In response, the attester queries the TPM, which generates a quote (a signed statement of the system's state) using its conventional cryptographic key. Before transmitting the quote, the attester adds an extra layer of security by signing the quote with a new post-quantum key, ensuring resilience against future threats posed by quantum computing. Once the verifier receives the attestation data, it validates the system's integrity by checking both the TPM-generated quote and the post-quantum signature. By implementing this hybrid attestation approach, we maintain compatibility with the existing TPM infrastructure while protecting the attestation process against quantum threats. This guarantees security even if classical cryptographic methods become obsolete. The thesis will explore the challenges and implementations needed to make attestation processes quantum-safe, with particular emphasis on extending the attestation agent and trust manager to integrate cryptographic mechanisms resistant to quantum attacks.

Chapter 2

Post-quantum cryptography and standardization of the algorithms

This chapter explores post-quantum cryptography, looking at the basics and advances that ensure cryptographic systems stay secure against quantum computers. In particular, there are two solutions against this threat: mathematical-based cryptographic solutions, which highlight their strength against quantum attacks, and Quantum Key Distribution (QKD), which explains its principles and practical uses as a key part of quantum-resistant security. The chapter also reviews the efforts of the National Institute of Standards and Technology (NIST) to standardize post-quantum cryptographic algorithms. It highlights the importance of a unified and secure cryptographic framework for future digital infrastructure, providing a clear understanding of the strategies and innovations shaping the evolution of cryptography in the quantum era. This discussion will highlight the integration of secure algorithms, particularly those used in this thesis.

2.1 Post-quantum cryptography

Post-quantum cryptography (PQC) refers to cryptographic methods that assume an attacker has a large-scale quantum computer. Also known as quantum-safe cryptography, it includes cryptographic algorithms considered quantum-safe, meaning they have not been proven vulnerable to cryptanalytic attacks by a quantum computer. Additionally, these systems should be capable of interoperating with existing communication protocols and networks. Post-quantum cryptography is a highly active area of research and development: the primary goal is to develop algorithms that can replace existing cryptographic protocols, ensuring the long-term security of sensitive information. Although large-scale quantum computers are not yet a reality, it is essential to focus on PQC now to prepare for future advancements. Some aspects should be considered:

1. longevity: PQC algorithms are designed to protect against classical and quantum-based attacks. This ensures their security even as the power of classical computing increases.

2. preparation: the development of PQC algorithms enables us to prepare for the future threat posed by quantum computers. Cryptographic systems are often used to protect data for extended periods, making it crucial to start planning for the possibility of quantum-based attacks now.

3. adoption: development, testing, and adoption of cryptographic systems require significant time. By starting the transition to PQC algorithms now, we can ensure that appropriate replacements for current cryptographic systems will be available when needed.

There is a risk that switching from pre-quantum cryptography to a post-quantum cryptosystem will damage security, not only failing to protect against quantum computers but also losing

protection against today's computers. To address this risk, it could be reasonable to deploy post-quantum cryptography as an extra layer with pre-quantum cryptography, rather than deploying it as a replacement for pre-quantum cryptography. In this way, we can guarantee one of the most important principles of Security, known as Security in depth [10]: if the enemy can defeat the first line of defense, there must be a second line to stop the attacker. A developer (or a normal user) does not have to rely on just one defense, as that defense may have a bug or problem. It is better to have multiple levels of defense because as the attacker breaks through the defenses, it will become increasingly difficult to continue penetrating.

Although double encryption and double signing sound straightforward, one must be careful with the details, because even the best algorithms and security mechanisms can be compromised by weak implementations. Coding errors, misconfigurations, or the use of insecure libraries can introduce vulnerabilities.

- In case of double encryption [11], if an encryption algorithm is vulnerable to cryptographic analysis attacks (such as a known-plaintext attack or brute-force attack), double encryption does not necessarily provide additional security. For example, if both encryption layers use the same algorithm with weak keys, an attacker might be able to break both encryptions with a reasonable amount of computational resources.
- In case of double signature [12], if the two digital signatures are generated using the same algorithm and key, compromising one key might compromise both signatures.

In a nutshell, it is dangerous to reuse keys across multiple systems or to assume that the presence of one component will compensate for weaknesses in another component: each component should be designed to be secure by itself.

The initial significant research on post-quantum cryptography started in the late 1990s. One of the first proposed PQC algorithms was the McEliece cryptosystem [13], introduced by Robert McEliece in 1978. This system is based on the theory of error-correcting codes and is known for its resistance to quantum-based attacks. It is a public key encryption scheme and uses a linear error-correcting code, typically a binary Goppa code[14], to encode messages, where the public key consists of a generator matrix for the code, obscured by a random permutation, while the private key is the original Goppa code and the permutation. Encryption involves adding a random error vector to the code word, and decryption leverages the private key to correct these errors and recover the plaintext. However, it has not seen widespread adoption due to its large key sizes.

2.1.1 Quantum safety and the Mosca inequality

For industries focused on protecting sensitive information from adversaries, it is crucial to be proactive in their approach to information security. Specifically, as described by M. Mosca in [15], it is necessary to consider:

- Shelf-Life Time (X Years): Number of years you need your cryptographic key to remain secure and your data to be protected.
- Migration Time (Y Years): Number of years required to develop, deploy, and migrate to the Quantum-Safe solution.
- Threat Timeline (Z Years): The number of years before large-scale quantum computers will be built, which can break the current cryptography algorithms.

Named after Michele Mosca, the Mosca inequality outlines the pressing timeline for the transition to quantum-safe cryptographic systems. As shown in 2.1, if a large-scale quantum computer (Z) is built before the infrastructure has been retooled to be quantum-safe and the required duration of information security has passed $(x + y)$, the encrypted information will not be secure, leaving it vulnerable to adversarial attack. In other words, if $X + Y > Z$ it becomes a serious concern, as the organisations will not be able to protect their assets for the required years from

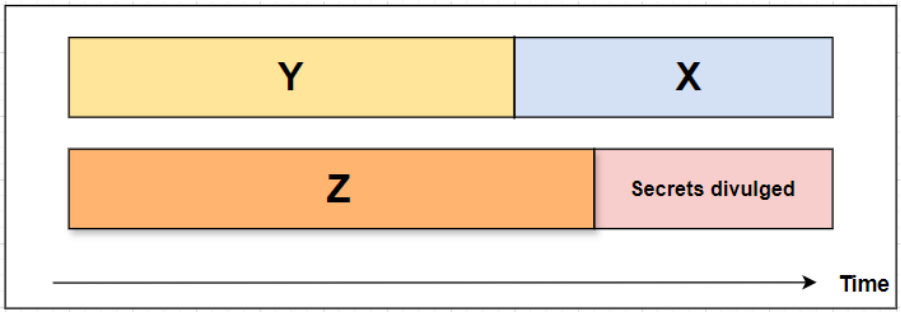

Figure 2.1. Mosca inequality

quantum attack. The threat is not just a concern for the future, but also a real danger in the present. Many hackers might be intercepting and obtaining the encrypted messages they will be able to decrypt when quantum computing resources become available in the future.

While quantum computers do exist today, they remain highly rudimentary and imperfect. The primary challenges in quantum computing are centered around creating high-precision hardware. Even with qubits capable of performing basic operations with a 0.1% error rate, these errors propagate and grow exponentially across the system, thereby limiting the practical size of a useful quantum computer. Each additional qubit doubles the power of a quantum computer; thus, when Google AI Quantum announced quantum supremacy in late 2019, their experiment utilized a processor with only 53 qubits.

2.1.2 PQC transition timeline

This perspective makes a set of recommendations to organisations on the process and timeline by which the PQC transition should take place [16]. Figure 2.2 presents a timeline of important PQC-related events that are expected to occur. This timeline is composed of three parallel sequences of events. The red timeline captures the two most important quantum threats and when they become of critical importance. The first one has already been introduced in section 1.1 and is known as a store-now-decrypt-later attack. It is already an active threat, corresponding to adversaries capturing valuable encrypted information now, storing it, and decrypting it later once large and fault-tolerant (LFT) quantum computers are available. The second quantum threat refers to the capability of breaking RSA and elliptic curve cryptography, the two most widespread public key algorithms for encrypting information today that can be broken with Shor's algorithm.

The gray timeline depicts the two actions required by organisations in transitioning to PQC. The first regards the strategic planning and technological experimentation for this transition, whereas the second regards the effective adoption of PQC in production systems. It is emphasized that the strategic planning phase must be completed well before LFT quantum computers can effectively attack RSA and ECC. Finally, the blue timeline concerns the standardisation processes organized by relevant government and industrial bodies, with particular focus on the National Institute of Standards and Technology PQC process to determine the fundamental security of proposed PQC candidates.

2.1.3 Timing the migration to PQC

When discussing quantum attacks, it is natural to question when the transition to post-quantum cryptography should begin. Although LFT quantum computers are not yet available, this section provides arguments emphasizing the importance of starting the PQC transition now:

- store now decrypt later: this issue has already been discussed in general terms; crucial trade secrets, medical records, national security documents, and other sensitive information have multidecade shelf lives and must remain confidential for extended periods. Consequently,

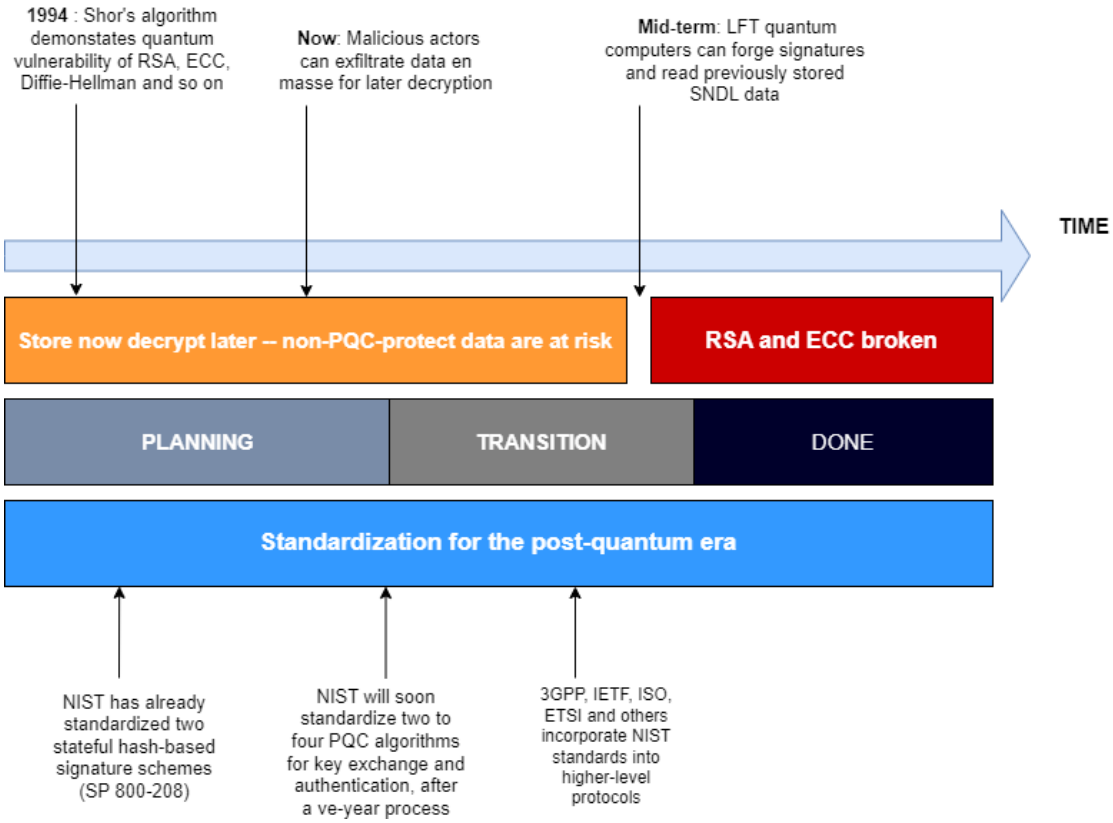

Figure 2.2. PQC transition Timeline

the SNDL attack is one of the most important reasons to avoid delaying the transition any longer.

- far-horizon projects: another reason for the immediate importance of PQC involves projects currently being designed and planned with long lifespans, often spanning multiple decades. Vehicles are a prime example: many cars, planes, trains, and ships in production today are expected to remain in service for up to 20 or even 30 years. This is especially relevant when application-specific hardware is used to implement cryptography, as it typically remains unchanged for the product's entire lifespan. Critical national infrastructure projects also exemplify this issue, where high availability is essential, and upgrading cryptographic software or hardware would incur unacceptable costs.

- cryptography transition takes time: Elliptic Curve Cryptography was proposed in the 1980s but, despite its greater efficiency in terms of space and speed compared to RSA (depending on parameterisation), it took over two decades to achieve widespread adoption. Hash functions also illustrate this prolonged adoption process. For instance, the NIST SHA-3 competition was announced in 2007, the winner was declared in 2012, yet even by 2021, SHA-3 had not seen widespread adoption. Therefore, cryptographic transitions, even simpler ones than the PQC transition, typically take several years or even decades. The PQC transition is particularly complex because many approaches are relatively new, and the performance of many candidates is significantly worse than current algorithms.

2.2 Technologies and Solutions for Post-Quantum Security

The following sections describe some of the most promising approaches to counter quantum threats. The first approach, quantum key distribution, relies on the laws of quantum physics to create a secure and theoretically impenetrable key distribution method.

On the other hand, mathematical solutions have been developed to provide robust security against decryption attempts by quantum computers. These methods are characterized by mathematical problems that are hard to solve, even for the most powerful quantum algorithms, making them an essential component for ensuring long-term security in the post-quantum landscape.

2.2.1 Quantum key distribution

Quantum Key Distribution addresses the challenge of securely exchanging a cryptographic key between two parties over an insecure channel. QKD relies on the fundamental principles of quantum mechanics, which are immune to the increasing computational power of classical computers. It can be carried out using the quantum properties of light, lasers, fiber optics, and even free-space transmission technology. There are two QKD protocols: BB84 (1984) and E91 (1991).

On a practical level, it has a few setbacks though, the range of communication is limited. So far, the best result on fiber optics was 12 km in December 2020 by the Indian Defence Research and Development Organisation.
Research has led to the development of many new QKD protocols exploiting mainly two different properties described right below.

- Prepare-and-measure (P&M) protocols use the Heisenberg uncertainty principle, which states that the measuring act of a quantum state changes that state in some way. This makes it difficult for an attacker to eavesdrop on a communication channel without leaving any trace. In the case of eavesdropping, the legitimate parties involved in the exchange can discard the compromised information and determine the amount of data that has been intercepted. This property was used in the BB84 protocol [17].
- Entanglement-based (EB) protocols use pairs of entangled objects shared between two parties. As explained in 1.2.1, entanglement is a quantum physical phenomenon that links two or more objects together so they have to be considered as one object afterward. Additionally, measuring one of the objects would affect the other as well. When an entangled pair of objects is shared between two legitimate exchange parties, anyone intercepting either object will alter the overall system. This would reveal the presence of an attacker along with the amount of information the attacker retrieved. This property was exploited in the E91 protocol.

BB84

The BB84 protocol [18] is a method for generating and distributing cryptographic keys using principles of quantum mechanics. BB84 exploits the polarisation of light to create a random sequence of qubits (keys) transmitted through a quantum channel.

BB84 uses two different bases: base 1 is polarized 0°(horizontal) or 90°(vertical), with 0°equal to 0 and 90°equal to 1. Base 2 is polarized 45°or 135°with 45°equal to 1 and 135°equal to 0. Alice (the sender) begins by sending a photon in one of the two bases, having a value of 0 or 1. Both the base and the value should be chosen randomly. Next, Bob (the receiver) selects the base 1 or 2 and measures a value without knowing which base Alice has used. The key exchange process continues until they have generated enough bits. Furthermore, Bob tells Alice the sequence of the bases he used but not the values he measured, and Alice informs Bob whether the chosen bases were right or wrong. If the base is right, Alice and Bob have equal bits, whereas if it is wrong the bits are discarded. In addition, any bits that did not make it to the destination are discarded by Alice. Now, Alice can use the key they just exchanged to encode the message and send it to Bob.

2.2.2 Mathematical solutions

There are many alternative mathematical problems to those used in RSA, DH, and ECDSA. The most researched mathematical-based implementations are the following: Lattice-based cryptography, Multivariate-based cryptography, Hash-based signatures, Code-based cryptography, and Isogeny-based cryptography.

Lattice-based Cryptography

Lattice-based encryption is the most popular algorithm for public key generation researched for PQC. A lattice is a network of infinitely many points; each vector is a point, and the set of vectors representing any point in the lattice is called a basis. In lattice-based encryption, messages are presented under vectors, and the public key is a matrix in which the messages are multiplied to generate the ciphertext. This is a form of public key cryptography that avoids the weaknesses of RSA. Rather than multiplying primes, lattice-based encryption schemes involve multiplying matrices. There are 3 lattice-based schemes:

- lattice-based encryption
- lattice-based signature schemes
- lattice-based key exchanges

Lattice-based cryptographic constructions are based on the hardness of lattice problems, and the core problem among all lattice problems is named the Shortest Vector Problem (SVP): the goal is to find the shortest non-zero vector within the lattice. This problem is NP-hard, and unlike the factorisation problem or the discrete logarithm problem, there is no known quantum algorithm to solve SVP with the help of a quantum computer. Lattice problems also benefit from something called worst-case-to-average-case reduction, which means that all keys are as hard to break in the easiest case as in the worst case when setting up any of the parameters of a lattice-based cryptosystem.

One of the most important in this family is NTRU, which is used for both encryption (NTRUEncrypt) and digital signature (NTRUSign) schemes. NTRU relies on the difficulty of factoring certain polynomials, making it resistant against Shor's algorithm. To provide a 128-bit post-quantum security level, NTRU demands 12881-bit keys. So far no known attacks have been successful against NTRU. Among all the lattice-based candidates mentioned above, NTRU is the most efficient and secure algorithm, making it a promising candidate for the post-quantum era. Until now, lattice-based cryptography is believed to be secure against classical and quantum-based attacks [4].

Multivariate-based Cryptography

Multivariate cryptography is another approach to PQC, based on the difficulty of solving systems equations. Multivariate cryptography is known for its small key sizes but is vulnerable to certain types of attacks. The security of this public key scheme relies on the difficulty of solving systems of multivariate polynomials over finite fields. Solving the multivariable polynomial problem has been proven to be nondeterministic polynomial-time (NP)-complete, making it well-suited for post-quantum cryptography (PQC). Research [19] has shown that developing an encryption algorithm based on multivariate equations is difficult. Multivariate cryptosystems can be used for both encryption and digital signatures. They are known to produce the shortest digital signatures among post-quantum cryptography (PQC) algorithms. The most promising signature schemes include Unbalanced Oil and Vinegar (multivariate quadratic equations), and Rainbow. UOV and Rainbow are SingleField schemes, meaning that all computations are performed over a single finite field. Rainbow is more efficient by using smaller digital signatures and key sizes.

Hash-based Cryptography

Hash-based cryptography [20] is a type of PQC based on hash functions, which are non-reversible functions that take a string of any length as input and produce a fixed-length output. Hash-based cryptography has the advantage of being relatively easy to implement and does not require large key sizes. However, there are concerns about the efficiency of hash-based schemes. Parameter b defines the desired security level of the system. For the 128-bit b security level, a secure hash

function is needed that takes an arbitrary length input and produces a 256-bit output. Therefore, SHA-256 is considered an optimal solution that fits well with the message m.

The security of such one-time signature schemes is based on the collision resistance of the chosen cryptographic hash function. Since hash function signatures cannot be used more than once securely, they are combined with structures like binary trees (Merkle tree) so that instead of using a signing key for a single one-time use signature, a key may be used for several signatures bounded by the size of the binary tree. Each position in the tree is calculated to be the hash of the concatenation of its children nodes. These nodes are computed successively, with the root of the tree being the public key of the global signature scheme. The leaves of the tree are built from one-time signature verification keys. In a nutshell, the hash-based signature (HBS) is a collection of many one-time signature (OTS) schemes. This idea was introduced by Merkle in 1979 and suffered some efficiency drawbacks, such as large signature sizes and slow signature generation. A significant strength of hash-based signature schemes is their flexibility, as they can be used with any secure hashing function, so if a flaw is discovered in a secure hashing function, a hash-based signature scheme just needs to switch to a new and secure hash function to remain effective. An important drawback of Merkle-related schemes is their statefulness: the signer must keep track of which one-time signature keys have already been used. This can be tricky in large-scale environments.

Currently, two hash-based signature schemes are under evaluation for standardisation. Specifically, the eXtended Merkle Signature Scheme (XMSS) which is a stateful signature scheme, and Stateless Practical Hash-based Incredibly Nice Collision-resilient Signatures (SPHINCS), which is as the name indicates a stateless signature scheme.

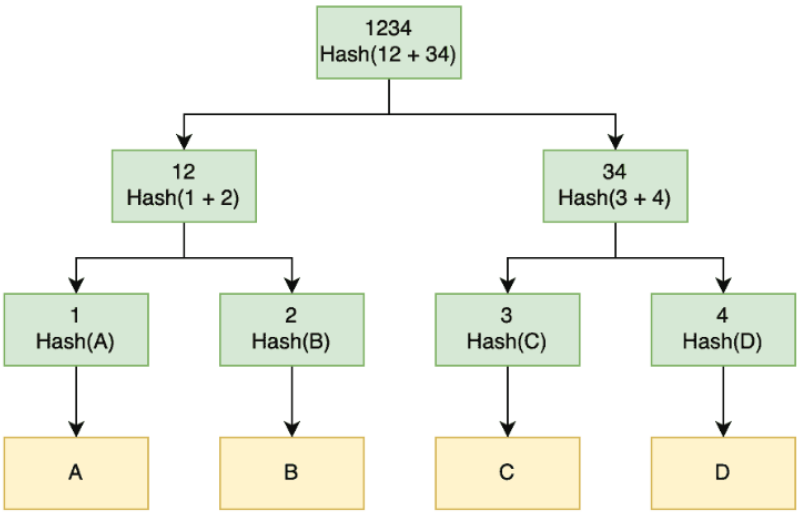

Figure 2.3. Merkle tree construction

Figure 2.3 is a symbolic example of a Merkle tree. Hashes typically have a fixed size, regardless of the amount of input data. This means that a Merkle Tree with a known number of leaf nodes will have a known size, regardless of the actual size of the data being hashed. The construction process of a Merkle Tree starts by splitting data into blocks. The leaf nodes of the tree are then the hashes of these blocks. Then all the other nodes in the tree are calculated by combining their immediate branches and generating a hash of them. In this example, there are four data blocks: “A”, “B”, “C”, and “D”. The first thing to do is calculate into hashes “1”, “2”, “3”, and “4”. Then, hash “12” is generated by combining hashes “1” and “2” and then hashing this result. It is repeated until the reach of the root node.

Code-based Cryptography

Code-based cryptography is another approach to PQC that relies on the security of error correcting codes. The algorithms are based on the difficulty of decoding linear codes and are considered robust to quantum attacks when the key sizes are increased by a factor of 4. In McEliece's idea, the sender adds a specific amount of random noise to encrypt the message. McEliece's cryptosystem features fast encryption and decryption processes with very low complexity, but makes use of large public keys (100 kilobytes to several megabytes). This technique is computationally hard to reverse using either a conventional or quantum computer, it is based on a mathematical problem called syndrome decoding which is known to be an NP-complete problem if the number of errors is unbounded.

An example of efficient error-correcting codes is the Goppa codes [14], which can be turned into a secure coding scheme by keeping the encoding and decoding functions secret and only publicly communicating a disguised encoding function that allows the mapping of a plaintext message to a scrambled set of code words. The secret mapping can be removed only in possession of the secret decoding function, to recover the plaintext [21].

A simple but slow attack strategy against McEliece's system is "information-set decoding" (ISD). An information set is a collection of codeword positions that determines the rest of the codeword. ISD guesses an information set, hoping that the ciphertext is error-free in those positions. What makes ISD slow is that, for large matrices, the ciphertext is extremely unlikely to be error-free.

The main practical challenge with these systems is their large key size. Many newer code-based systems introduce additional structure into public keys to enable greater compression, but some of these proposals have been successfully broken.

Isogeny-based Cryptography

Cryptosystems of this type rely on the property of congruent graphs of elliptic curves over finite fields to create a secure system. The security of Isogeny cryptography is based on so-called supersingular isogeny problems, or in other words, finding the isogeny mapping between two supersingular elliptic curves with the same number of points. Several specific schemes are based on this, such as the CSIDH (Commutative Supersingular Isogeny Diffie-Hellman) key exchange scheme [22], an alternative quantum attack candidate to the Diffie-Hellman key exchange scheme. An isogeny-based public key encryption, SIKE, is a PKE/KEM selected after the third round of NIST, and there are some studies on this scheme; however, this PKE/KEM was attacked and removed from the list of fourth-round candidate algorithms.

	Usage	Advantages	Disadvantages	Most Important
Code-based encryption	Encryption/Decryption Signature	Secure, very fast encryption, short ciphertext	Large public keys	NTRU
Lattice-based	Encryption/Decryption Ket exchange Signature	Short ciphertext and keys, very fast encryption	Complexity	McEliece
Isogeny-based	Key exchange	Difficult of computing isogenies between supersingular elliptic curves	Complexity	SIDH
Multivariate-based	Signature	Very short signatures	Diffficulty of solving that kind of systems	Rainbow
Hash-based	Signature	High security , simple decription	Management of state	SPHINCS

Figure 2.4. Post-quantum algorithms

2.3 PQC standardisation

Several standardisation bodies are working on the standardisation of PQC. These efforts are being led by NIST (based in the US), the International Standards Organisation (ISO), the Internet Engineering Task Force (IETF), and the European Telecommunications Standards Institute (ETSI). Each of these bodies is at a different stage in the process and is focusing on different PQC schemes.

2.3.1 Standardisation of stateful hash-based signatures

Stateful hash-based signatures (HBS) are digital signature schemes whose security relies solely on the security of hash functions. This represents an advantage compared with other digital signature schemes [16]. Furthermore, hash functions are among the most studied topics in cryptology, meaning that their security properties are well understood, including their expected resistance against quantum attacks. The statefulness property means that the signer must keep track of a state between signature generations. The state is an increasing counter, and reusing the same state would compromise the security of the system. There are stateless HBS schemes too, but those are comparatively less efficient than their stateful counterparts. Given their optimal security properties and acceptable performance metrics, stateful HBS have already been (or will soon be) standardized by multiple standardisation bodies, being the first PQC standards available for widespread adoption. NIST is running two standardisation efforts related to PQC, one of which is discussed in section 2.3.2 below, whereas the other (completed) was focused specifically on stateful HBS and has already finished: NIST standardized the same schemes for which the IETF published RFCs, namely the XMSS [23] and LMS [24] schemes, and their multitree variants.

2.3.2 NIST standardisation

The advent of quantum computers and quantum cryptographic algorithms has threatened the existence of ciphers based on mathematical difficulty. Therefore, in 2016, the US National Institute of Standards and Technology launched a competition to find algorithms that resist the power of quantum computing. Currently, the NIST standardisation process is in its fourth round [25]. NIST, with the involvement of the cryptographic community, has begun to develop a minimum set of acceptance and evaluation criteria for potential candidates. Comparing such varied approaches brings unique challenges, such as weighing up security, key sizes, latency, bandwidth, and ease of secure implementation. The process considers three cryptographic functionalities: stateless digital signature, asymmetric encryption, and key encapsulation mechanisms. Parameter sets for five security levels, ranging from the equivalent of an exhaustive key search on AES 128 (level I) to AES 256 (level V), are analyzed. This allows cryptosystems from different families to be roughly compared with each other.

2.3.3 First round

The first round of NIST's PQC standardisation process began in December 2016 and ended in July 2019. During this period, NIST received 82 submissions, which were judged on their security, performance, and implementation characteristics. NIST established criteria for evaluating algorithms, including security for classical and quantum computers, flexibility, and ease of implementation. NIST also conducted several rounds of testing and analysis to ensure the algorithms met these criteria. At the end of the first round, NIST selected 26 candidate algorithms for further research and evaluation in the second round [25].

2.3.4 Second round

The second round of the PQC standardisation process began in January 2019 and ended in July 2020. During this period, the focus was on evaluating and analyzing the 26 candidate algorithms selected from the first round. Submissions were evaluated based on their security against different

types of attacks, their speed and memory consumption performance, and flexibility in terms of key size and security level. The second round involved extensive testing and evaluation of candidate algorithms, including software and hardware implementations. Based on the second round assessment results, NIST selected 15 candidates who advanced to the third round of the standardized process.

2.3.5 Third round

The third round started in July 2020 and ran for 18 months, with 15 candidates selected after the end of the second round. In this round, NIST asked the candidates to analyze the proposals and prove they achieved adequate security in experiments and theory. After 18 months of selection, 7 out of 15 candidate algorithms were selected and classified as "finalists" (4 asymmetric encryption or key encapsulation mechanisms (KEMs) and 3 stateless signature schemes) and 8 were classified as 'alternatives' (5 asymmetric encryption or KEMs and 3 stateless signature schemes). The selected candidate algorithms are suitable for most applications ready for standardisation, and alternative candidate algorithms are potential candidates for the future. Finalists include:

- PKE/KEM algorithms: Classic McEliece, CRYSTALS-Kyber, NTRU, Saber;
- Digital signature schemes: CRYSTALS-Dilithium, Falcon, Rainbow.

Alternative candidate algorithms include:

- PKE/KEM algorithms: BIKE, FrodoKEM, HQC, NTRUPrime, SIKE;
- Digital Signature schemes: GeMSS, Picnic, SPHINCS+.

During this evaluation, the Rainbow digital signature algorithm was broken by Ward Beullens [26] using a classical computer. After the selection process, NIST selected four algorithms to standardize right after the third round, including:

- The PKE/KEM algorithm CRYSTALS-Kyber;
- Digital signatures are CRYSTALS-Dilithium, Falcon, and SPHINCS+.

The intention of keeping alternative candidates in the process is explained by several reasons, including achieving diversity of primitives and suitability to special use cases.

2.3.6 Fourth round

In the fourth round, during the selection process, the SIKE candidate algorithm was broken. After four rounds of selection, three rounds have been completed, and the final round is currently underway. Three candidate algorithms remain for PKE/KEM standardisation. Out of 69 valid candidates in the first round, only four PKE/KEM algorithms and three digital signature schemes have been selected.

2.3.7 Analysis of the finalist candidates

It is important to underline that here are considered only some members of the corresponding families, those having "5" as security level: taking Kyber as an example, the Kyber family includes variants such as KYBER512, KYBER768, and KYBER1024, which primarily differ in the following aspects:

- Security level

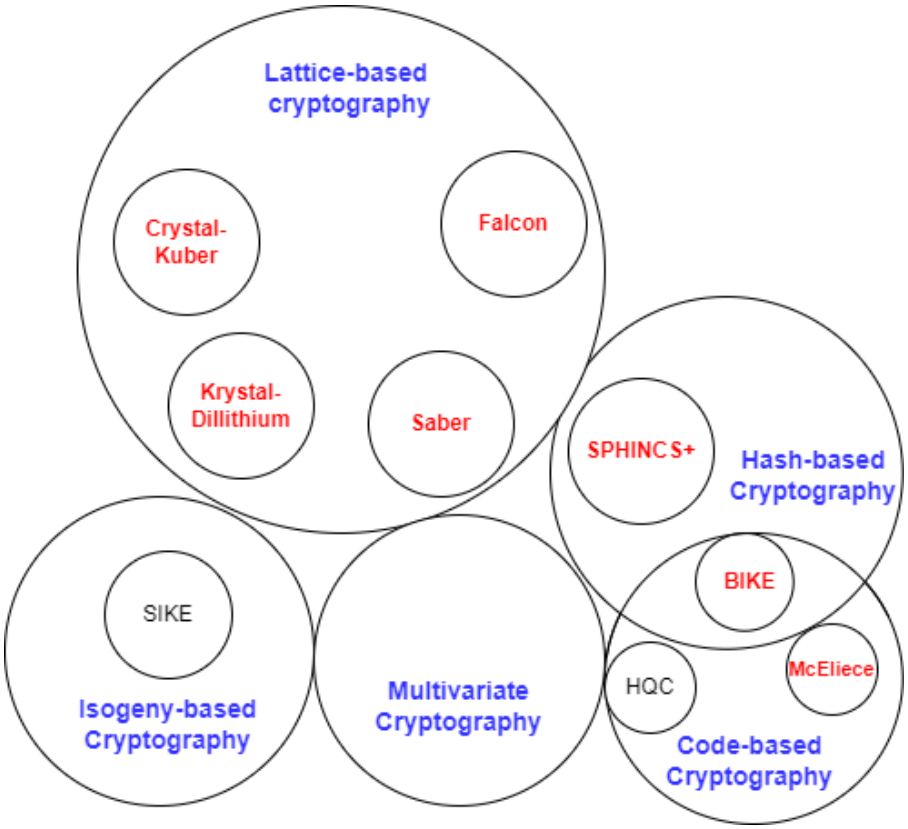

Figure 2.5. Standardisation of PQC algorithms [25]

- Key sizes and other parameters
- Computational Efficiency
- Resource Usage

In conclusion, in the case of Kyber, the choice between KYBER512, KYBER768, and KYBER1024 will depend on the desired balance between security level and available computational resources. Variants with higher security levels are suitable for scenarios where security is an absolute priority, while those with lower security levels offer better performance in terms of computational efficiency and resource usage.

2.3.8 Research Status on PQC

As indicated in [25], studies focus on candidate algorithms selected through the stages. The Venn diagram in Figure 2.5 briefly describes the current state of research on PQC, in which the circles with blue words are the basis of encryption, while the red words are the names of the algorithms that are candidates. NIST last update was on August 13, 2024, by announcing the selection of three finalist algorithms for post-quantum cryptographic standards:

- FIPS 203: The Module-Lattice-Based Key-Encapsulation Mechanism Standard, which is based on the CRYSTALS-Kyber algorithm.
- FIPS 204: The Module-Lattice-Based Digital Signature Standard, derived from the CRYSTALS-Dilithium algorithm.
- FIPS 205: The Stateless Hash-Based Digital Signature Standard, derived from SPHINCS+.

2.4 SPHINCS+

SPHINCS+ is a stateless hash-based signature scheme submitted to the NIST post-quantum crypto project. The parameters listed in table 2.1 can be modified to create instantiations of the signature scheme with different levels of security, signature size, and computational complexity. As an example, SPHINCS+ can be instantiated with three different hash functions:

- SPHINCS+-SHAKE256
- SPHINCS+-SHA-256
- SPHINCS+-Haraka

These signature schemes are obtained by instantiating the SPHINCS+ construction with SHAKE256, SHA-256, and Haraka, respectively. The stateless nature of SPHINCS+ distinguishes it from earlier hash-based schemes, which often required maintaining a state to track key usage. By eliminating the need for state management, SPHINCS+ reduces implementation complexity and minimizes the risk of key misuse, making it more suitable for real-world applications. In the implementation phase of this thesis, SPHINCS+ will be utilized to demonstrate its practicality and effectiveness in providing postquantum secure digital signatures.

Liboqs is an open-source C library that provides implementations of post-quantum cryptographic algorithms. It has been developed under the Open Quantum Safe (OQS) project. In liboqs, the SPHINCS+ algorithms are categorized with suffixes “s” and “f” to denote different parameter sets:

- “s”: Represents the “small” parameter set, optimized for smaller signature sizes.
- “f”: Represents the “fast” parameter set, optimized for faster signing and verification times.

This classification is independent of the “simple” and “robust” variants related to tweakable hash functions in SPHINCS+.

2.4.1 Difference Between SPHINCS and SPHINCS+

SPHINCS+ is an improved version of the original SPHINCS digital signature scheme, addressing key limitations in its predecessor. The differences between SPHINCS and SPHINCS+ are as follows:

- Improved Efficiency: SPHINCS+ incorporates several optimisations that reduce the computational cost and improve the overall efficiency of the signing process. For example, it reduces the number of hash-function calls required for generating and verifying signatures.
- Smaller Signature Sizes: One of the major criticisms of SPHINCS was its large signature size. SPHINCS+ introduces improved techniques such as ForsTree and Haraka, which help to achieve more compact signatures while maintaining security.
- Flexible Parameter Sets: SPHINCS+ provides multiple parameter sets, allowing users to balance signature size, speed, and security requirements.

2.4.2 SHAKE-256

SHAKE-256 is a member of the SHA-3 family of cryptographic hash functions standardized by NIST in FIPS 202. SHAKE stands for the Secure Hash Algorithm KECCAK, and the number 256 refers to its security strength and flexibility in output size. Unlike traditional hash functions such as SHA-2, SHAKE-256 is an extendable output function (XOF), which means it can produce an output of arbitrary length. The most important properties of this algorithm are:

Parameter	Value
Hash function	$H_{\text{alg}} \in \{\text{SHA-256, Shake256, Haraka}\}$
Security parameter	$n \in \{128, 192, 256\}$
Height FORS trees	t
Number of FORS trees	k
Winternitz parameter	$w \in \{4, 16, 256\}$ bits
Height of the hypertree	h
Number of subtrees	d
Tweakable hash	{robust, simple}

Table 2.1. SPHINCS+ instance parameters

- Extendable Output: SHAKE-256 differs from fixed-length hash functions because it allows the output length to be specified based on application requirements.
- Security Strength: SHAKE-256 achieves a security level of 256 bits against collision resistance and pre-image resistance for outputs of sufficient length. Specifically:
  - Collision resistance: 2^{128}
  - Preimage resistance: 2^{256}

  This makes it suitable for cryptographic applications that require high security, such as digital signatures, key derivation, and message authentication.

In the implementation of this work, both the `SIG_sphincs_shake_256s_simple` and `SIG_sphincs_shake_256` variants, representing the simple and fast versions of the algorithm, have been compared during the testing phase to evaluate performance differences in terms of signing time, verification time, signature size, etc.

2.5 Dilithium

Dilithium is a post-quantum cryptographic algorithm based on the hardness of mathematical problems related to lattices, such as the "Short Integer Solution (SIS) problem" and the "Learning With Errors (LWE) problem", which have proven to be resistant even against the most advanced attacks.

The operation of Dilithium is structured into three main phases:

- key generation; specific polynomials and matrices are chosen to form the public and private keys.
- message signing; when a user wants to sign a message, they perform a series of mathematical operations on these polynomials to produce a signature
- signature verification

Unlike other post-quantum algorithms, it offers high performance in terms of key generation and signature verification speed, while keeping the size of keys and signatures relatively small. Compared to Falcon, which employs more complex mathematics to achieve smaller signatures, Dilithium is easier to implement and more resistant to certain types of attacks. When compared to SPHINCS+, which relies on hash functions rather than lattices, Dilithium is significantly faster. Crystals-Dilithium has 6 variations :Dilithium2, Dilithium3, Dilithium5, Dilithum2_AES,

Dilithium3_AES and Dilithium5_AES. Variations Dilithium2, Dilithium3, Dilithium5 are on the size of the polynomial matrix, secret key range, and masking vector coefficient range. Dilithium_AES version uses AES-256 in counter mode instead of SHAKE-128 or 256 to expand the matrix and the masking vectors and to sample the secret polynomials.

In the implementation of this work, the `SIG_alg_dilithium_5` variant has been used during the testing phase to evaluate performance differences in terms of signing time, verification time, signature size, etc.

2.6 Mayo

The Oil and Vinegar signature scheme, proposed in 1997 by Patarin, is one of the oldest and best-understood multivariate quadratic signature schemes. It has excellent performance and signature sizes but suffers from large key sizes on the order of 50 KB, which makes it less practical as a general-purpose signature scheme. To solve this problem, it has been proposed MAYO, a variant of the UOV signature scheme whose public keys are two orders of magnitude smaller. MAYO makes possible to represent the public key very compactly. The usual UOV signing algorithm fails if the oil space is too small, but MAYO works around this problem by "whipping up" the base oil and vinegar map into a larger map, that does have a sufficiently large oil space. In the context of the post-quantum cryptography standardisation process promoted by NIST, MAYO was presented as a candidate to become a security standard. However, during the evaluation phases, concerns arose regarding the robustness of MAYO. In particular, some experts raised doubts about its long-term security, suggesting that it might not provide the necessary protection against advanced attacks. As a result, MAYO was not selected among the final algorithms standardized by NIST. Instead, NIST chose other algorithms considered more reliable and secure for protecting information in the era of quantum computers. Although MAYO was not selected as a final NIST standard, analyzing its performance provides insights into potential optimisations for future MQ-based schemes. It helps determine whether MQ cryptography can be further refined for practical use cases.

In the implementation of this work, the `SIG_alg_mayo_5` variant has been used during the testing phase to evaluate performance differences in terms of signing time, verification time, signature size, etc.

Chapter 3

Trusted computing and remote attestation

As technology advances and more devices connect to networks, ensuring their security becomes increasingly important. In a world where systems are always communicating with each other, it's essential to keep data safe and ensure devices can be trusted. Trusted Computing is a key solution to address these challenges. It focuses on creating secure environments to protect sensitive information, even when dealing with systems that may not be fully trustworthy. One of its main methods is remote attestation, where a host verifies its hardware and software configuration to a remote host. Through Remote Attestation, a verifier can obtain guarantees about the integrity of the system and the security operations of the attester. This chapter provides an overview of Trusted Computing, focusing on remote attestation. This discussion will set the stage for exploring the integration of liboqs to enhance the cryptographic mechanisms used in Keylime, particularly in attestation workflows.

3.1 Trusted computing

Attackers often target systems at the lowest level, such as the operating system (OS). If the OS is compromised, other security measures, like firewalls or anti-malware, become ineffective. Physical access to the system can also allow attackers to boot an alternative OS, especially with network boot options. To prevent such attacks, both the boot system and the OS must be protected. In the past, systems used BIOS (Basic Input Output System), but today it has been replaced by UEFI, which includes native firmware signature support. UEFI checks the firmware's integrity before allowing the bootloader to verify the OS. To further enhance security, an external hardware root of trust can be used to validate the BIOS, ensuring the process is secure and unaltered. The external crypto chip handles the BIOS validation, removing reliance on the CPU. Today, the goal is to minimize the trust required in the system to make it easier to verify and audit.

A trusted component or platform behaves as expected, but that does not mean it is completely secure. It simply means that no modifications have been made to what was originally programmed. To achieve that, an attestation is performed, which is evidence of the current state of the platform that can be verified by someone else. Attestation can be carried out in two ways: hardware attestation and software attestation. Hardware attestation relies on a secure, tamper-resistant module (a TPM or a secure element) to generate cryptographic evidence, providing a higher level of security. Software attestation, on the other hand, verifies the system state through software measurements, but is generally considered less secure because it lacks the physical protection of hardware-based solutions.

3.1.1 Trusted Computing Base (TCB)

There is significant experimental evidence showing that conventional computer systems are not secure. Moreover, fixing security flaws as they are discovered has proven to be an insufficient approach to achieving truly secure systems. The only sound approach to the provision of secure computer systems is to design security into those systems right from the start. Generally speaking, small, simple, and localized mechanisms are easier to correct than large and complex ones. Therefore, the first task in designing a secure system is to find a way to structure it so that its security mechanisms are as small and as simple as possible.

The Evaluation Criteria define the totality of security mechanisms within a secure system as its Trusted Computing Base (TCB), which encompasses all the hardware, software, and firmware components critical to the system's security. These components are essential for enforcing and maintaining the system's overall security policy, ensuring the integrity and confidentiality of sensitive operations. TCB is a collection of system resources (hardware and software) responsible for maintaining the security policy of the system. An important feature is its ability to prevent being compromised by any hardware or software that is not part of the TCB.

3.1.2 Root of Trust

A RoT is an essential security component that provides a set of functions that the rest of the device or system can use to establish strong levels of security. Sitting outside the system software, a RoT initiates a chain of trust by ensuring the computer only starts the boot process after confirming no malicious code is present. It ensures any software running on the device is trustworthy.

The Root of Trust is a crucial component that must always behave as expected, as any malfunction cannot be detected. It serves as the foundation for the establishment of trust in a platform, being a hardware device with known behavior verified by a certificate cite rta. There are different RoTs in a trusted computing environment:

- Root of Trust for Measurement (RTM): measurement means computing values that tell whether a system is good. It is a trusted implementation of a hash algorithm, responsible for the first measurement on the platform. Measures and sends its integrity measurements to the RTS (another RoT). Usually, the CPU executes the CRTM (Core Root of Trust for Measurement) software component.
- Root of Trust for Storage (RTS) : is a special portion of memory that is shielded/secured. Shielded means that no other entities but the CRTM can modify the value.
- Root of Trust for Reporting (RTR): an entity that securely reports the content of RTS.

Essentially, there is the RTM that computes the measurement. This measurement is then securely stored in the RTS. When needed, the RTR requests the stored measurement and provides it to an external verifier. The TPM typically includes both the RTS and the RTR: it is a secure storage and a trusted component for reporting. The Core Root of Trust for Measurement is still required, which is why the TPM is used together with a secure boot. This ensures the system correctly installs the CRTM, which continuously measures and sends results to the hardware component.

3.1.3 Chain of trust

In general, there is a component A that measures component B, and once these measures have been performed, A stores the result of them in the RTS 3.1.2. Then component B will do the same tasks with another component C, storing the results. Then, with all those measures, it is possible to ask RTR for the measurement stored by B and C from the RTS. If component A is trustworthy, then the verifier knows if B and C are good because the expected hash value is known (if it is equal is good, it fails otherwise). An important example is the SSL/TLS internet security

Figure 3.1. Chain of trust

standard, which is based on a trust relationship model, also called the "certificate chain of trust." X.509 digital certificates validate the identity of a website, organization, or server and provide a trusted platform for the user to connect and share information securely [27]. SSL/TLS internet-based Public Key Infrastructure (PKI) allows users to exchange data using public and private key pairs, obtained and exchanged by a trusted certificate authority (CA). This reputable entity is responsible for issuing, retaining, and revoking public key certificates over insecure networks.

When visiting a website via a secure connection, the site sends a digital certificate to your browser. Your internet browser compares the issuer with a list of trusted Certificate Authorities (Root CA). If a match cannot be found, the client browser checks to see whether a trusted Root CA signs the issuing CA certificate. The browser's chaining engine continues verifying the issuer of each certificate until it finds a trusted root or upon reaching the end of the trust chain, as shown in figure 3.2.

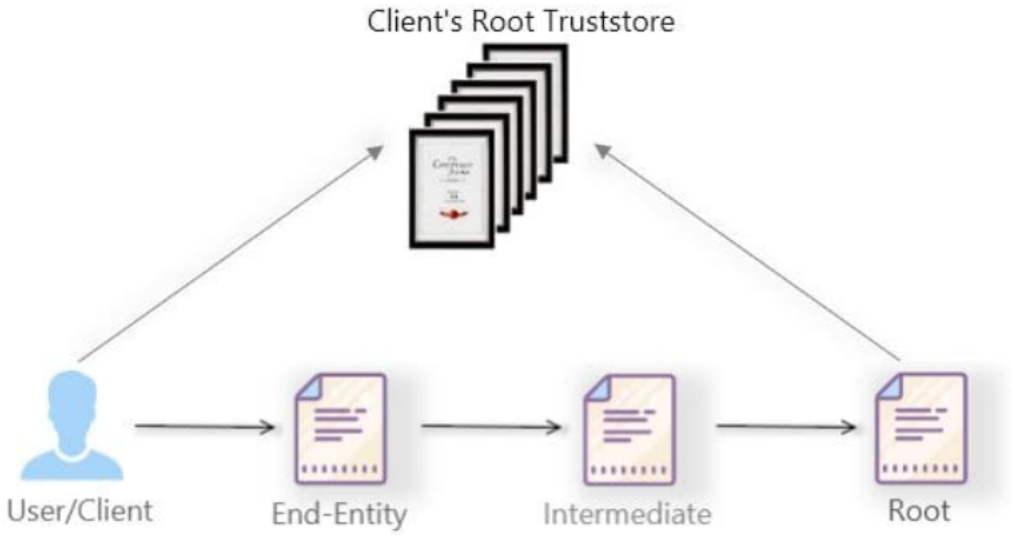

Figure 3.2. Chain of trust certification

The chain of trust certification aims to prove that a particular certificate originates from a trusted source. If the certificate is legitimate and links back to a Root CA in the client browser's Truststore, the user will know that the website is secure based on interface trust indicators. Otherwise, if the chain of trust fails verification, a certificate can not prove its validity on its own, and the browser will warn the user of a potential security risk.

3.2 TPM

3.2.1 TPM overview

The Trusted Platform Module is a security module that delivers the basis of a safe computing environment [28]. It is a cheap component, less than one dollar usually, and is available on most servers, laptops, and PC. Processes that need to keep secrets, such as digital signing, can be made more secure with a TPM. Mission-critical applications requiring greater security, such as secure email or secure document management, can offer a greater level of protection when using a TPM. With a TPM, it is easier to ensure that the artifacts needed to sign secure email messages have not been tampered with by software attacks. Attestation or any other TPM functions do not transmit personal information of the platform's user.

TPM is tamper-resistant, but not tamper-proof. Tamper-proof means it cannot be attacked, while tamper-resistant means it tries to resist various kinds of tampering. Although TPM contains cryptographic modules, it is not a high-speed cryptographic engine. Unlike smart cards, the TPM is bounded to a specific platform[28]. The TSG (Technical Skill Group) defines trusted computing as the expectation that a device will behave in a particular manner for a specific purpose. The purpose of the TPM is to provide this assurance to the client and the users interacting with the client. It is certified Common Criteria EAL4+, which is quite a high level (The "+" sign indicates that additional security requirements have been included beyond the standard ones for EAL4, making the certification more rigorous and including further controls or tests against specific threats). It is a passive component, meaning it does not take control of the computer but must be managed by the CPU.

The main building blocks of TPM are specified by Trusted Computing Group as shown in figure 3.3.

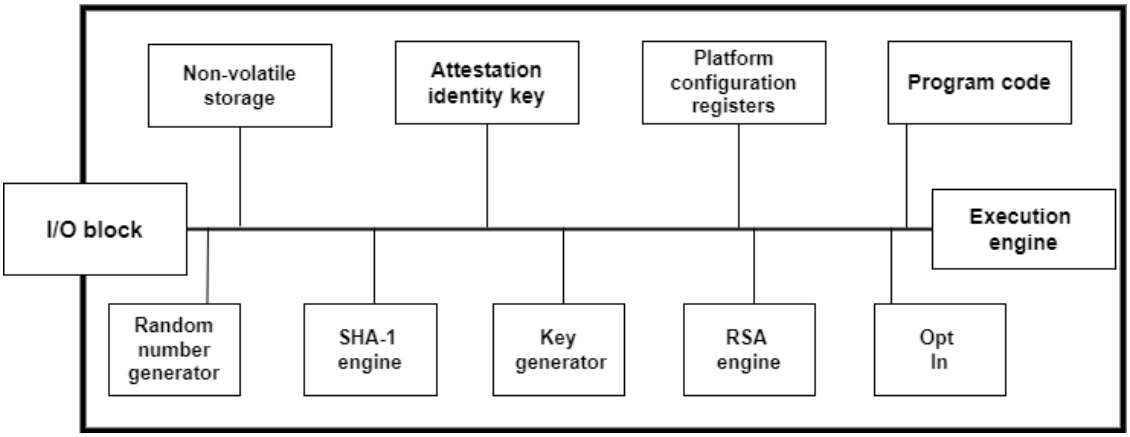

Figure 3.3. TPM generic building blocks

The I/O block acts as a bridge between internal and external components. It not only manages information flow between components via the bus but also enforces access policies for various components.

Trusted Computing Group and interoperability

The Trusted Computing Group (TCG) is an international de facto standards body of approximately 140 companies engaged in creating specifications that define PC TPMs, trusted modules for other devices, trusted infrastructure requirements, and protocols necessary to operate into a trusted environment. Without standard security procedures and shared specifications, it is not possible for components of the trusted environment to interoperate, and trusted computing applications cannot be implemented to work on all platforms. From a cryptographic perspective, trusted modules must be able to use the same algorithms. While standard algorithms may have

weaknesses, they are thoroughly tested and gradually replaced or improved when vulnerabilities are found [29].

3.2.2 TPM features

The TPM is a chip that authenticates the hardware itself. Since most attacks originate from unknown hardware, being able to identify the device eliminates the possibility of someone stealing a key and reusing it later on [28]. But this is not the only benefit of TPMs. TPMs provide an entire cryptographic suite of tools:

- program code contains the firmware used to initialize the device.
- the execution engine executes the program code that performs initialization and measurement taking.
- it contains a hardware random number generator (not a pseudo RNG). The random numbers produced are used to generate cryptographic keys, nonces, and strengthen passwords.
- secure generation of cryptographic keys for limited uses (especially with the RSA algorithm).
- TPM can be used for remote attestation: it is used to store the hash summary of the hardware and software configuration.
- TPM can perform binding (data encrypted using the TPM bind key, a specific key inside the component, cannot be decrypted outside that specific TPM, because it is a unique RSA key derived from a storage key). It is a good solution because even if the data is stolen, there is no way to decrypt it. It is also a bad solution because if it is needed to export data to move to another machine, a complex procedure is required.
- it is an additional level of security, in which not only is data encrypted with a key that is internal to the TPM, but as part of the decryption operation, the operation requires the TPM state to be the same as when the data was encrypted (known as sealing or bounding). It is important to highlight that the state is the collection of all the applications running on the platform, together with the configuration files.
- it is a non-volatile storage: it stores several long-term keys and authentication credentials such as Endorsement Key (EK), Storage Root Key (SRK), owner's password, and persistent flags. The SRK naturally is the root of this secure storage and thus manages it. The EK is a unique feature in TPM: for a TPM to operate, the EK pair must be embedded in it; the private key is permanently embedded in it (i.e., unique to each TPM and thus the platform). The public key is stored in a certificate, and it is only used in a limited number of operations. EK is used to generate an alias, the Attestation Identity Keys (AIKs), used for routine operations. The EK pair is generally provided by the manufacturers before shipping [30].

Opt-In

As the name implies, the user has to opt-in to use the TPM, which means to take ownership and configure the TPM. In the process of taking ownership, the TPM will transition through several states as described below. Changing the state of these flags requires authorization. The opt-in block ultimately provides mechanisms and protection to maintain the TPM state via the state of these flags [28]. In short, the TPM can exist in a range of states from disabled (and deactivated) to fully enabled and ready for ownership.

- disabled/enabled: all operations are restricted except reporting TPM capabilities and accepting updates to the PCRs 3.2.6.
- activated/deactivated: the subtle difference between this operational variable and the previous one is that when TPM is deactivated, it will respond to a change of state or owner.

- owned/unowned: in an owned state, a key pair has been established, and the owner can perform any operations on TPM, including state change.

Some BIOSes also provide a "Clear" option for the TPM. Clearing the TPM erases the Storage Root Key and the owner, making all keys and data useless. This command is normally used before transferring the machine to a new owner.

3.2.3 TPM 1.2

There are two important versions of the TPM, 1.2 and 2.0. The first one was the most widely used until a few years ago, and it is rather inflexible because it contains:

- fixed set of algorithms.
- one root key, named Storage Root Key.
- one storage hierarchy for the platform user.
- sealing (having data being able to be decrypted only with a certain state) was tied to PCR value 3.2.6, where PCR are special registers inside the TPM.

TPM 1.2 supports a single "owner" authorization, with an $RSA2048$ Endorsement Key for signing/attestation and a single $RSA2048$ Storage Root Key for encryption (used to protect other keys and data). This means a single user or entity ("owner") has control over both the signing/attestation and encryption functions of the TPM. In general, the SRK serves as the parent for any keys created in TPM 1.2.

3.2.4 TPM 2.0

TPM 2.0 is a big improvement because it provides cryptographic agility. For backward compatibility, it continues to have SHA-1, but it also offers SHA-256. TPM 2.0 has the same functionality that is represented by the EK for signing/attestation and SRK for encryption as in 1.2 3.2.3, but the control is split into different hierarchies:

- Platform Hierarchy: for data coming from outside. Platform means the system, which is hosting the TPM. It contains non-volatile storage, keys, and data related to the platform. It is used for maintenance functions.
- Storage Hierarchy: for internal storage. It is used by the privacy administrator for storing keys and data related to privacy.
- Endorsement Hierarchy: used for generating cryptographic evidence that can be trusted by external entities. It is responsible for creating and managing keys used in attestation and signature processes, ensuring that the system's identity and integrity can be verified by outside parties.
- Null Hierarchy, a stateless hierarchy with no persistent storage. It is mainly used for operations that do not require long-term key storage or when temporary keys are needed. This hierarchy provides flexibility for tasks that don't need to rely on permanent keys, such as specific cryptographic tasks only valid for the current session.

In TPM 2.0, the new Platform Hierarchy is intended to be used by platform manufacturers. The Storage and Endorsement hierarchies and the Null hierarchy will be used by the operating systems and OS-present applications. Each hierarchy has a dedicated authorization (password as a minimum) and a policy. Keys of the various hierarchies are unrelated. Each hierarchy has its own unique "owner" for authorization. Because of this, TPM 2.0 supports four authorizations, which would be analogous to the single TPM 1.2 "owner".

3.2.5 TPM objects

These are the objects that are managed by the TPM:

- Primary keys: in particular, endorsement keys and storage keys. They are derived from one of the primary seeds. They can be recreated using the same parameters, assuming the primary seed has not been changed.
- Keys and sealed data objects (SDO): they are protected by a Storage Parent Key (SPK). Randomness for these keys come from the TPM RNG, which is internal to the TPM.

An object inside the TPM has two parts: the public and private areas. In contrast, the sensitive area is external. While the public and private areas are mandatory, the sensitive area is optional.

- Public area: used to uniquely identify an object.
- Private area: contains the object's secrets and exists only inside the TPM.
- Sensitive area: encrypted private area used for storage outside the TPM. Not compulsory.

3.2.6 TPM Platform Configuration Register (PCR)

The TPM can report the current state of the system. To store and report it the TPM contains a special set of registers named PCR. They are registers that keep the history of the platform configuration. One strict requirement is that these registers must be reset at the system restart or whenever there is a power loss (all the registers will start with 0)[28]. This reset ensures that the PCRs start in a clean state and can accurately store fresh measurements of the system's integrity. The reason for this reset is to guarantee that, if the system is reconfigured or rebooted, the PCRs will contain the most up-to-date integrity metrics, reflecting the current state of the system and preventing old, potentially outdated data from influencing security decisions. These registers support only two operations: reset and extend. The extend operation is:

$$PCR_{\text{new}} = \text{hash}(PCR_{\text{old}} \| \text{digest\_of\_new\_data})$$

The old value contained in the PCR, concatenated to the digest of some new data, is hashed. The result becomes the new value of the PCR. The standards require TPMs to have at least 16 PCRs of size 20 bytes. Registers 0-7 are reserved for TPM use. Registers 8-15 are free for any application's use, including operating system [30]. In other words, the PCRs contain system measurements: the TPM gives the possibility to read directly these values, but it is not considered a trustworthy operation. What is done instead is an operation called quote: a quote is a signed report from the TPM, that contains the current PCR values, and it also uses a nonce, to prove that the quote is fresh (it is referred to the register values of the present, not of the past). Quotes can be provided to other parties for PCR verification (remote attestation).

3.3 Remote attestation

Until now, there has been no comparison between the values in the PCRs and the expected values, as this comparison would be performed locally. However, if the system has been manipulated, the comparison could be inaccurate. This makes it challenging to use a TPM for self-monitoring. Instead, the TPM must rely on an external party, which is the purpose of remote attestation.

3.3.1 Remote attestation procedures

Remote attestation is the process of asserting the properties of a target by providing evidence to an appraiser over a network[31]. Attestation is considered very important nowadays: systems that have been attested and verified to be in a good state (for some value of "good") can improve overall system posture. Viceversa, systems that cannot be attested and verified to be in a good state can be given reduced access or privileges, taken out of service, or flagged for repair. For example:

- A bank backend system might refuse to transact with another system that is not known to be in a good state.
- A healthcare system might refuse to transmit electronic healthcare records to a system that is not known to be in a good state [32].

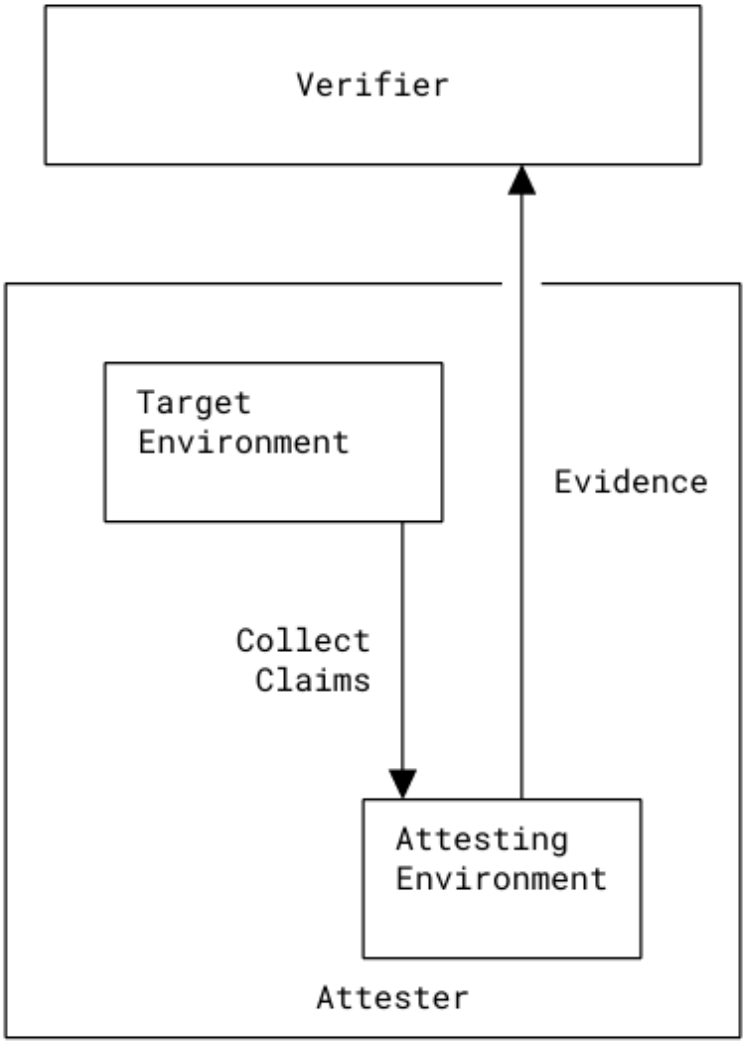

Figure 3.4. Environment with an Attester [32]

In remote attestation, an Attester consists of at least one Attesting Environment and at least one Target Environment co-located in one entity. Other implementations might have multiple Attesting and Target Environments. Claims are collected from Target Environments. Attesting

Environments collect the values and information to be represented in claims by reading system registers and variables, calling into subsystems, and taking measurements on code, memory, or other relevant assets of the Target Environment. Attesting Environments then format the Claims appropriately; typically, they use key material and cryptographic functions, such as signing or cipher algorithms, to generate evidence[32]. The Attester signs the collected measurements using a secure private key, usually stored in a hardware component like the TPM or the Trusted Execution Environment. This ensures the integrity and authenticity of the measurements. In the case of a TPM, it does not actively collect Claims itself. Instead, it requires another component to feed various values to the TPM. Thus, in this case, an Attesting Environment would be the combination of the TPM and the component providing it with the measurements.

It is important to consider that an entity can take on multiple Remote Attestation Procedures (RATS) roles (e.g., Attester, Verifier, Relying Party, etc.) at the same time. Multiple entities can cooperate to implement a single RATS role as well. In a nutshell, the combination of roles and entities can be arbitrary.

The remote verifier performs validation in two steps:

1. First, it verifies the signature cryptographically: crypto + ID. There is also an ID, as a table must exist linking each node identifier to its corresponding public key. For example, if a challenge is sent to a device with ID1, the response will be signed with that key.

2. Then, the verifier uses the corresponding public key to verify the signature and ensure that the measurements have not been tampered with. It will compare the measurements against Reference Measurements, also known as golden values, which are the possible values that are already known [32]. If the measurements match, the verifier concludes that the attester is in a trustworthy state.

3.3.2 Trust model

This section analyzes one of the possible communication models between the key entities of a trust model. It is called the Passport model and involves the attester, Verifier, and Relying Party. The Passport Model is so named because of its resemblance to how nations issue passports to their citizens. The nature of the evidence an individual needs to provide to its local authority is specific to the country involved. Thus, in this immigration desk analogy, the citizen is the Attester, the passport-issuing agency is a Verifier, and the passport application and identifying information (e.g., birth certificate) is the Evidence [32]. Figure 3.5 shows a data flow diagram for their communication. The Attester conveys its Evidence to the Verifier for appraisal and the Relying Party receives the Attestation Result from the Verifier.

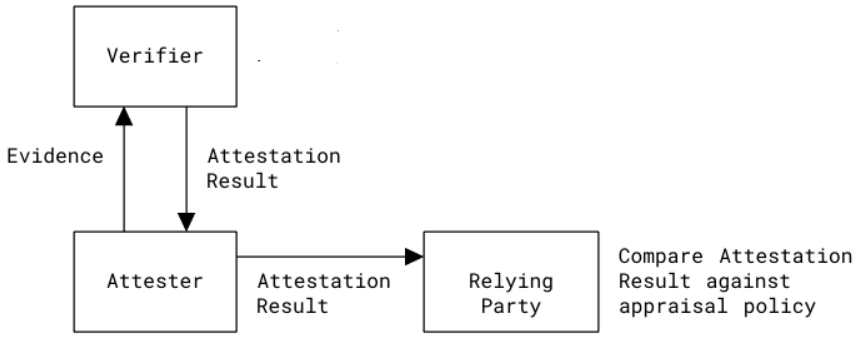

Figure 3.5. Passport model [32]

- Relying party: a Relying Party trusts a Verifier that can appraise the trustworthiness of information about an Attester. For a stronger level of security, the Relying Party might require that the Verifier first provide information about itself that the Relying Party can use to assess the trustworthiness of the Verifier before accepting its Attestation Results. Such trust is expressed by storing one or more "trust anchors" in a secure location known as a "trust anchor store". A trust anchor represents an authoritative entity via a public key and associated data.
- Attester: In some scenarios, evidence might contain sensitive information; an Attester must trust the entities it shares evidence with to ensure sensitive data is not disclosed to unauthorized parties [32]. When evidence contains sensitive information, an Attester typically requires that a Verifier authenticates itself (e.g., by establishing a TLS session).
- Verifier: the Verifier relies on the manufacturer or its hardware to assess the trustworthiness of the manufacturer's devices. Such trust is expressed by storing one or more trust anchors in the Verifier's trust anchor store.

3.3.3 Principles for attestation architectures

Five principles [31] are crucial for attestation architectures. While an ideal attestation architecture would satisfy all five, in real systems, only an approximation of the ideals is possible.

1. Fresh information: Assertions about the target should reflect the running system, rather than just disk images. Some measurement tools provide only start-up time information about the target, assuming that its security-relevant properties remain intact.
2. Attestation mechanisms should provide detailed information about the target, ensuring its full internal state is accessible to local measurement tools.
3. Constrained disclosure: a target should be able to enforce policies that determine which measurements are sent to each appraiser.
4. Semantic explicitness: The semantic content of attestations should be explicitly presented in logical form. The target's identity should be defined by these semantics, allowing an appraiser to collect attestations about it.
5. Trustworthy mechanism: Appraisers should receive evidence of the trustworthiness of the attestation mechanisms on which they rely. In particular, the attestation architecture in use should be identified to both the appraiser and the target.

3.3.4 Domain separation

Domain separation [31] is essential for establishing trust in attestations, particularly in ensuring the integrity of measurement tools. A measurement tool must be capable of delivering accurate results about a target of attestation, even if the target is compromised. This is an important consequence of Principle 5. First, it must have access to the target's state to distinguish whether that target is corrupted or uncorrupted. Second, the measurement tool's state must be inaccessible to the target, so that even if the target is corrupted, it cannot interfere with the results of the measurement. There are various ways to achieve this separation. One approach is to virtualize the target, allowing the measurement tool to run in a separate virtual machine from the target cite ravm.

Chapter 4

Keylime

Keylime is a TPM-based remote attestation and runtime integrity measurement solution that allows cloud users to monitor remote nodes using a hardware-based cryptographic root of trust. Originally developed by the security team at MIT's Lincoln Laboratory, Keylime is now maintained by the Keylime community. In this thesis, Keylime was chosen for its reliable TPM-based approach to remote attestation, ensuring the integrity of remote systems in untrusted environments. Keylime's flexibility and open-source nature made it an ideal choice for integrating additional security features. This chapter will explain Keylime's architecture, its key components, and the crucial phases of remote attestation, setting the foundation for the improvements discussed in the next sections.

4.1 Introduction

KeyLime is a technology used to centralize the remote attestation of distributed systems. In particular, KeyLime was introduced as the first end-to-end IaaS trusted cloud key management service that supports all the desirable features listed above:

- Secure Bootstrapping: the system should allow a tenant to securely provision an initial root secret into each of their cloud nodes. This root secret can then serve as a foundation for generating additional secrets, enabling advanced security services.
- Compatibility: the system should provide tenants with the capability to integrate hardware-rooted cryptographic keys into the software, strengthening the security of their existing services.
- Scalability: the system must scale efficiently to support provided features and services across numerous cloud nodes, given the dynamic nature of creating and removing IaaS resources.
- System Integrity Monitoring: the system should enable the tenant to monitor the integrity status of cloud nodes and detect any deviations, with a response time of less than one second.
- Secure Layering (Virtualization Support): the system should enable a tenant to perform secure bootstrapping and integrity monitoring within virtual machines using a TPM integrated into the provider's infrastructure. This process requires cooperation with the provider but should be implemented with minimal privileges granted to them.

To enable these functions, they relied on the TPM chips. TPM chips have been used to secure many different use cases, and one of their strengths is their ability to be a hardware root of trust for systems; it is really where KeyLime is using it as the integrity management hardware root of trust. Keylime's mission is to make TPM Technology easily accessible to developers and users, without the need for a deep understanding of the lower levels of a TPM's operations. Keylime contains four main components: the Verifier, the Registrar, and the Agent.

- The Verifier continuously verifies the integrity state of the machine where the agent is running on.
- The Registrar is a database of all agents registered with Keylime and hosts the public keys of the TPM vendors.
- The Agent is deployed on the remote machine that needs to be measured or provisioned with secrets, which are stored in an encrypted payload released once trust is established.
- The Tenant is a command-line management tool that allows the user/administrator to control the agents.

Users typically perceive a system as trustworthy when interacting with cloud or iOS platforms, assuming the service company has taken all necessary steps to ensure security. This trust is often implicit, based on the belief that the provider has conducted thorough due diligence. However, true trust in such systems must be built from the ground up, starting with the hardware and extending through the entire software stack. Maintaining system integrity becomes a significant challenge in large-scale environments such as data centers. When malware compromises the system, administrators cannot manually check every machine for signs of malicious activity. The issue becomes even more complicated when malware targets firmware, where detection is much more difficult.

4.2 Main components

Figure 4.1 provides a high-level overview of the Keylime architecture. It shows an agent running on each cloud node, communicating with the tenant system. Keylime supports multiple distribution scenarios: single node (multi-user), multi-node (Datacenter, IoT), Multi-tenant(Cloud), VMs, etc.

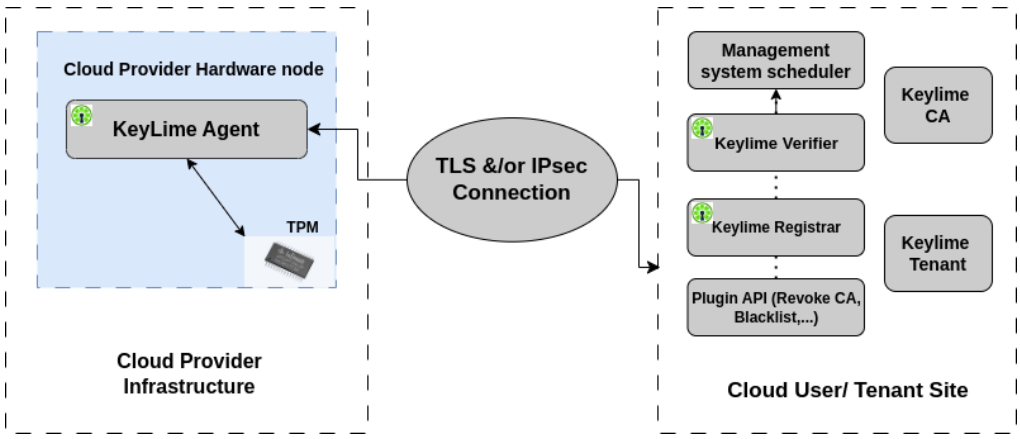

Figure 4.1. Main Components of Keylime

Keylime mainly consists of an agent, two server components (Verifier and Registrar), and a command line tool: the Tenant.

- Keylime Verifier: it continuously receives TPM reports from the agent. The verifier implements the actual attestation of an agent and sends revocation messages if an agent is not in the trusted state. Once an agent is registered for attestation (using the tenant or the API), the verifier continuously pulls the required attestation data from the agent. This can include a quote over the PCRs, the PCR values, the NK public key, the IMA log, and the UEFI event log. Afterward, the quote is validated, and additional data validation can be configured.

- Keylime Agent: it runs within the operating system being attested. Its primary role is to interact with the TPM to register the Attestation Key, generate quotes, and gather necessary data, such as UEFI and IMA event logs, to enable the system state attestation. The agent can also be configured to listen for revocation notifications sent by the verifier when a system's attestation fails. In such cases, a revocation message can trigger changes to the system's local policies, preventing the compromised device from accessing shared resources.
- Keylime Registrar: it is a centralized repository where each agent registers and reports its TPM public keys. This centralized approach eliminates the need for individual components to query each other for this information. Instead, the Registrar provides a single, well-defined location from which all relevant entities can reliably access the necessary data. The agent registers itself in the registrar. The registrar manages the agent enrollment process, which includes getting a UUID for the agent, collecting the EK_{pub}, EK certificate, and AK_{pub} from an agent, and verifying that the AK belongs to the EK. Once an agent has been registered with the registrar, it is ready to be enrolled for attestation. The tenant can use the EK certificate to verify the trustworthiness of the TPM.
- Keylime Tenant: the tenant is a command-line management tool shipped by Keylime to manage agents. This includes adding or removing the agent from attestation, validating the EK certificate against a certificate store, and retrieving the agent's status. It also provides the necessary tools for the payload mechanism and revocation actions.

4.3 Main phases

The Keylime framework can be subdivided into the following operational phases:

1. The Node Registration Protocol.
2. The Key Derivation Protocol.
3. The Continuous Remote attestation.
4. The Revocation Framework.

4.3.1 Node Registration Protocol

When the agent starts, it contacts the registrar to enroll the standard credentials of the TPM installed on the system. As represented in 4.2, the cloud agent sends to the registrar its $UUID$, along with the AIK_{pub}, the EK_{pub} and the EK_{cert} of the TPM. The registrar stores this information and challenges the cloud node to prove ownership of the EK_{priv} and AIK_{priv} corresponding to the public keys it received. The registrar creates the challenge in this way: it generates an ephemeral symmetric key K_e, it computes a hash of AIK_{pub} (denoted $H(AIK_{pub})$) and it encrypts this two information with EK_{pub}. When the cloud agent receives the registrar's challenge, it passes this encrypted blob to the ActivateIdentity TPM command. The TPM will correctly decipher K_e only if it owns EK_{priv} corresponding to EK_{pub} and AIK_{priv} corresponding to AIK_{pub}. The cloud agent proves it retrieved K_e by sending the $HMAC$ of its $UUID$ computed with K_e to the registrar. Upon receiving the response, the registrar recomputes the $HMAC_{Ke}$ ($UUID$) and, if the result is equal to the agent's response, it marks the cloud agent $UUID$ as active. It starts sending the node's TPM credentials when asked.

4.3.2 Bootstrap Key Derivation Protocol

The second step of the Keylime Framework involves the Three-Party Bootstrap Key Derivation Protocol. This protocol ensures the secure delivery of the bootstrap key K_b to the cloud node after verifying that it is in a trusted state. Initially, the tenant generates a symmetric encryption

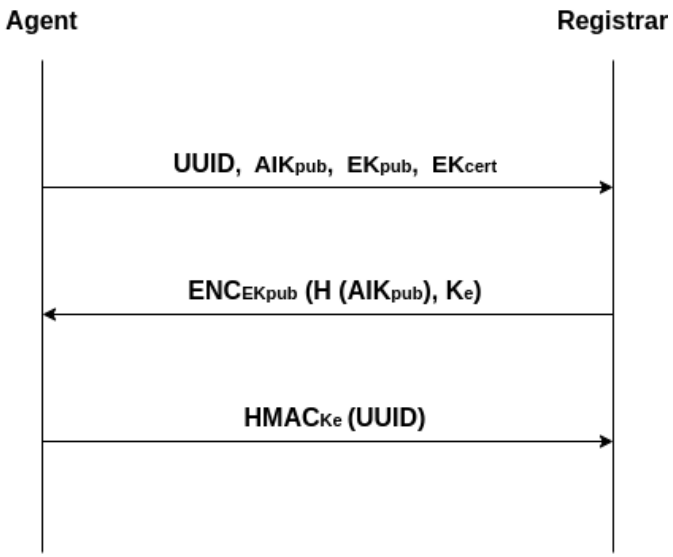

Figure 4.2. Agent Registration phase

key K_b and splits it into two parts U and V (random value) such that $U = K_b \oplus V$. The tenant sends U to the Cloud Node and shares V with the verifier, which will forward it to the cloud node only after a successful integrity verification. Once the tenant obtains the cloud node's $UUID$ and IP address, it notifies the verifier of the intent to initialize the node.

The Tenant connects to the Cloud Verifier over a secure channel, a mutually authenticated TLS, and provides V, the cloud agent's $UUID$, the IP, a TPM policy, and optionally it specifies a whitelist of acceptable PCR values and the MB refstate. In particular:

- The TPM policy specifies both the PCRs that the TPM quotes have to contain and the expected values associated with those PCRs;
- The whitelist is used for validating the IMA Measurement List and contains the list of trusted digests for the configuration files and the programs running on the cloud node
- The MB refstate is the Measured Boot reference state and is used for validating the Measured Boot ML.

At this point, both the tenant and Verifier can now attest the node in parallel. In the end, the cloud agent will receive U from the tenant and V from the CV. Since the cloud agent does not have a certified software identity key to establish secure communications, it generates an ephemeral asymmetric key NK and sends the public part of this key, NK_{pub}, to the CV and the tenant, so that they can use it to cipher V and U respectively and transmit them securely over an untrusted network. In particular, to prove the authenticity of NK_{pub}, the cloud Agent uses the 16th PCR value in the TPM quote to bind NK to the identity of the TPM. In this way, the identity of NK_{pub} is bound to the TPM identity and this allows the tenant and CV to authenticate NK by validating the TPM quote. Now the interactions will be examined in detail.

Interaction between the Cloud Verifier and The Agent

In this phase, the CV sends a request for a TPM quote to the cloud agent, specifying a fresh nonce and a mask that indicates the PCRs that the TPM quote has to contain. The node sends back NK_{pub} along with the quote $QuoteAIK(nonceCV, 16 : H(NK_{pub}), xi : yi)$, where $16 : H(NK_{pub})$ represents the PCR 16 containing the hash of NK_{pub}, while $xi : yi$ represents the PCRs requested by the CV with their respective values. Then, the CV asks the registrar for the node's TPM credentials (AIK_{pub},EK_{pub}, and EK_{cert}) over a server-authenticated TLS and uses AIK_{pub} to

verify the authenticity of the quote; if the quote is authentic, the CV verifies that the cloud node has a trusted state by comparing the PCRs values contained in the quote with the trusted values specified in the TPM policy and by validating the IMA ML with the whitelist. The CV also verifies that the received NK_{pub} is correct by computing the hash over NK_{pub} and checking that the result is equal to the content of PCR 16. If all the verification steps are passed, the CV sends back to the cloud agent the V share, encrypted with NK_{pub}, and then it starts the Continuous Remote Attestation phase described in the following section. Otherwise, the CV does not send V and sets the state associated with the cloud node as $INVALID\_QUOTE$.

Interaction between the Tenant and the Agent

The interactions of this phase occur in parallel to the one described above, and they are similar except for some small differences. The tenant requests a TPM quote to the cloud agent, specifying a fresh nonce and an empty PCR mask. The empty PCR mask is because, differently from the CV, the tenant does not use the quote to verify the trusted state of the node but only to verify the identity of the TPM to authenticate NK_{pub}. In this phase, the tenant verifies that:

1. The public key contained in EK_{cert} is equal to EK_{pub}.
2. The issuer of EK_{cert} is a trusted TPM manufacturer whose certificate is contained in a tenant's local repository (called "tpm_cert_store").
3. The signature of EK_{cert} is authentic, verifying it with the public key contained in the TPM manufacturer's certificate.

If one of the previous checks is not passed, the tenant notifies the CV, which sets the state of the cloud node to $TENANT\_FAILED$ and stops the periodic attestation on the cloud node. Instead, if the TPM of the cloud node is authentic, the tenant verifies the validity of the quote with the AIK_{pub} key, then it verifies the correctness of the received NK_{pub} in the same way as the CV does. If NK_{pub} results authentic, the tenant sends to the cloud agent the U share encrypted with NK_{pub}, the $HMAC$ over node's $UUID$ computed with $K_b(HMAC\_K_b(UUID))$ and the encrypted payload $Enc_{Kb}(d)$.

When the cloud agent receives U from the tenant or V from the CV, it verifies whether it has received both shares of K_b and, in this case, it computes $U = K_b \oplus V$. Then it checks if the derived K_b is the correct one by computing $HMAC_{K_b}(UUID)$ and verifying that it is equal to the value sent by the tenant; if so, the cloud agent uses K_b to decipher the encrypted payload and proceeds with the startup of the tenant's service. After decrypting the payload, the cloud agent deletes K_b and V while it stores U in the TPM NVRAM to automatically support the node's reboot or migration without the need for the tenant to interact again with the cloud node. Every time the cloud agent needs the V share after reboot or migration, it sends CV a new NK_{pub} along with the TPM quote; upon receiving a new NK_{pub}, the CV provides the V share to the cloud agent, so it can recombine K_b and decipher the encrypted payload again.

The TPM primitives used during the Three Party Bootstrap Key Derivation Protocol are:

- tpm2_pcrreset: Reset one or more PCR banks. More than one PCR index can be specified. The reset value is manufacturer-dependent and is either a sequence of 00 or FF on the length of the hash algorithm for each supported bank.
- tpm2_extend: Extends a PCR.
- tpm2_quote: Provide quote and signature for a given list of PCRs in given algorithm/banks.
- tpm2_checkquote: Uses the public portion of the provided key to validate a quote. generated by a TPM. This will validate the signature against the quote message and, if provided, verify that the qualifying data and PCR values match those in the quote.
- tpm2_nvdefine: Define an NV index with a given auth value.
- tpm2_nvwrite: Write data specified via FILE to a Non-Volatile (NV) index.

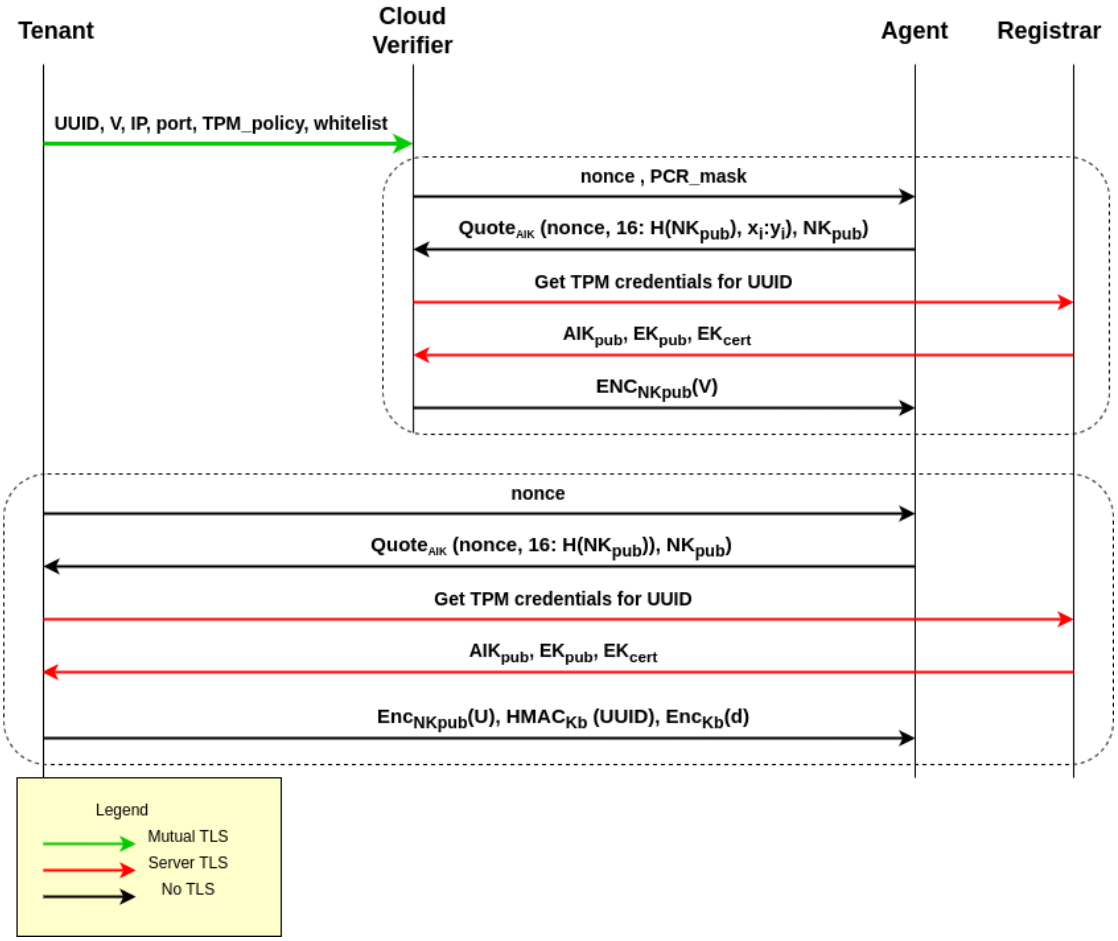

Figure 4.3. Bootstrap Key Derivation Protocol

[33]

4.3.3 Runtime Remote Attestation

Keylime provides a mechanism to continuously monitor the integrity of remote systems and verify their state during the boot process. After the Bootstrap Key Derivation Protocol is completed, the Keylime framework transitions into the third stage: Continuous Remote Attestation. During this phase, the framework ensures that the remote system has undergone a secure boot process, creating a trusted environment for application deployment. To maintain this trustworthiness over time, the CV routinely checks the integrity of the cloud node. This is achieved by leveraging the integrity measurements collected by IMA (Integrity Measurement Architecture) for the applications executed on the system.

As shown in Figure 4.4, the CV periodically sends requests to the cloud agent to retrieve updated Integrity Reports (IRs). In particular, the Verifier sends the nonce and a PCR mask. The agent uses the TPM to generate the quote. The quote includes:

- The nonce (to prove the quote is fresh).
- The requested PCR values.
- IMA Measurement List: Tracks hashes of loaded binaries and configuration files.
- Measured Boot Measurement List: captures the state of the system during the boot process.

The CV asks the registrar for the node's AIK_{pub} over a server-authenticated TLS and uses it to verify the authenticity of the quote. It validates these reports to detect any unauthorized

changes or integrity violations within the system. The validation process ensures that the system remains trusted by carefully analyzing the information provided in the IR. The verifier performs several checks in order to declare the quote as valid, it verifies that:

- the quote signature is valid, checking it with the AIK_{pub} key provided by the registrar;
- the quote contains all PCRs specified in the TPM policy;
- the quote contains PCR 16 and its content is equal to the digest (computed with the hash algorithm associated with the PCR 16) of the NK_{pub} sent by the cloud agent during the Bootstrap Key Derivation Protocol and stored in the local DB;
- the IMA ML matches the PCR 10 value;
- the measurement events contained in the IMA ML match the whitelist;
- the PCR values specified in the Measured Boot ML match the corresponding PCR values contained in the quote;
- the Measured Boot ML matches the MB refstate provided by the tenant.

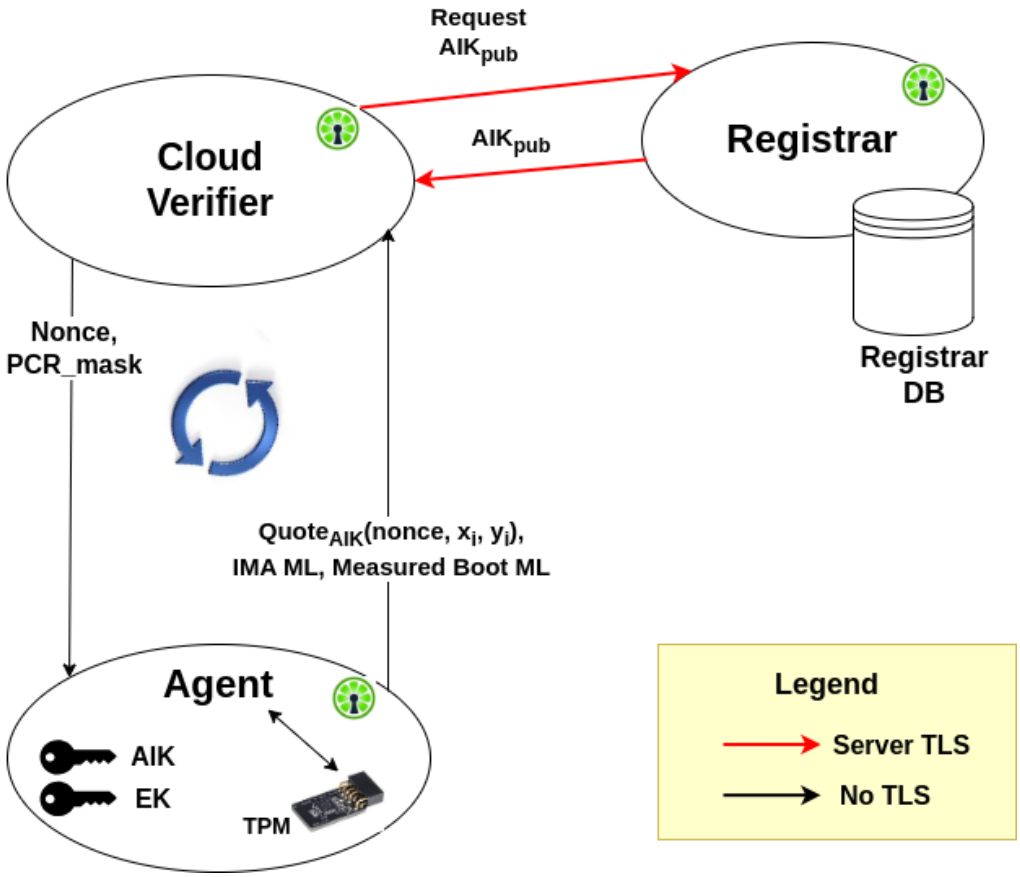

Figure 4.4. Runtime Integrity Monitoring

The time interval between consecutive attestation requests directly impacts the speed at which potential compromises are detected. By default, this interval is set to two seconds but can be adjusted in Keylime's configuration file. However, the minimum possible latency is restricted to at least 500 milliseconds, as the TPM quote operation typically requires more than 500 milliseconds.

4.3.4 Revocation framework

When the CV detects that a cloud node is untrusted, it triggers the Revocation Framework. This framework relies on the revocation notifier, a ZeroMQ server spawned by the CV at startup. This server implements the publish/subscribe pattern: the CV publishes a new revocation event by sending a signed message to the ZeroMQ server, which forwards it to all its subscribers. The subscribers can be:

- the software CA, which, upon receiving a revocation message for an untrusted cloud node, revokes the certificate of the identity key owned by that node and publishes an updated CRL;
- the cloud agents which, upon receiving a new revocation message, execute the "local action" scripts for ring-fencing the untrusted node;
- other tools that want to be notified about the events related to the trusted computing layer.

The presence of a file on the node is not sufficient to trigger any action. However, if the file begins running, the Verifier proceeds to disconnect the node. A possible scenario is shown in figure 4.5, where two cloud nodes establish an IPsec connection, each running a Keylime agent.

1. The Cloud Verifier performs the attestation loop on both nodes, checking that everything is good.
2. At some point, malware infects the Cloud Node 1, so the attestation fails and the verifier starts the revocation framework.
3. The Revocation Notifier sends a message signed with the revocation key to all subscribers(in this case, the cloud node 2).
4. The CA revokes the certificate for Cloud Node 1 and publishes a new CRL.
5. The node 2 can cut off its own IPsec connection with the affected node.

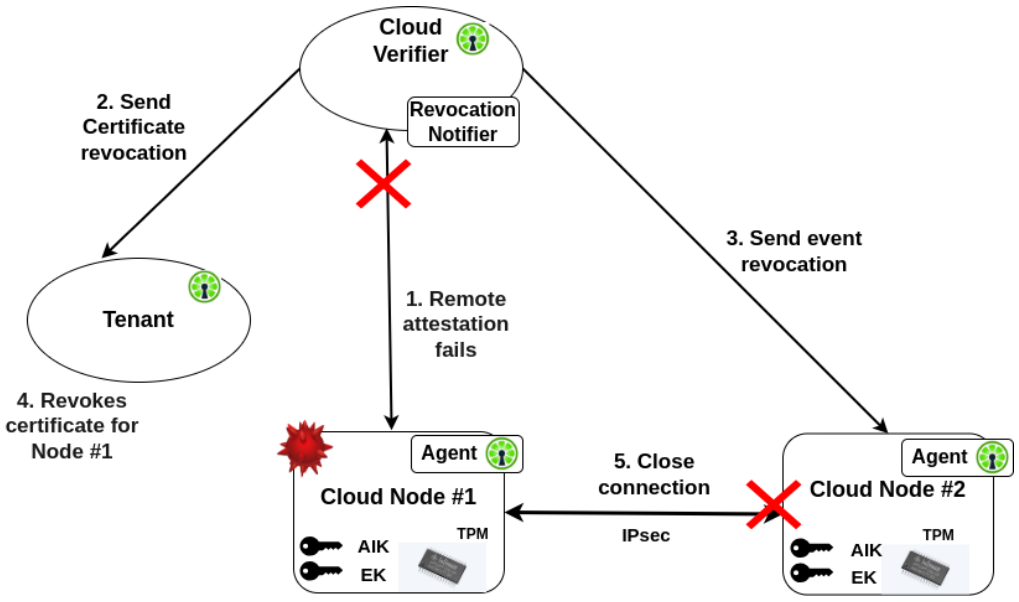

Figure 4.5. Revocation framework

4.4 Integrity Measurement Architecture (IMA) in Keylime

TCG's specifications define the core concepts of Trusted Computing, specifying the RoTs that a Trusted Platform must implement. IMA, the Linux implementation of the integrity measurement system outlined by TCG, extends the chain of trust from BIOS to the application layer. It measures all executables, configuration files, and kernel modules as they are loaded, storing these measurements in the TPM. This allows an external entity to verify not only the system's boot but also which software has been loaded, whether it is trustworthy, and whether its configuration is correct. Since, unlike the boot process, the order of software loading in an operating system is unpredictable, IMA records measurement events in a Measurement Log (ML) file and uses a PCR (typically PCR 10) to protect the integrity of the ML. An external entity can verify that the ML has not been tampered with and analyze each event to determine whether the system state change is trustworthy.

IMA enables a remote entity to confirm that an application running on a different system has an adequate level of integrity to be trusted. For a proper integrity check, the measurement list must be:

- Fresh: not vulnerable to replay attacks.
- Complete: containing all measurements taken up until the time of attestation.
- Unaltered: the measurements must remain intact without tampering.

4.4.1 Remote Attestation with IMA

The IMA measurement process begins when an IMA Hook in the attesting system receives a Measurement Event (ME), such as loading a binary or opening a file for reading or writing. The hook's role is to measure the received event, specifically by calculating the hash value of the file content using a secure hash function. The resulting file digest, along with other metadata, is stored in an ordered list of MEs within the kernel. The corresponding digest is then extended into a PCR in the TPM (typically PCR 10) using the extend operation. This measurement process allows the Verifier to check the integrity state of the attesting system by initiating a Remote Attestation. The challenge begins when the Verifier sends a request to the Agent, including a nonce. Upon receiving the request, the Attestation Agent asks the TPM for a quote, which includes the nonce from the challenger and the PCR values (these typically include the PCRs related to the boot process and PCR 10), which contain the IMA measurements aggregate. The TPM then returns the quote, signed with the $AIKpriv$. Next, the Attestation Agent retrieves the IMA Measurement Log (ML), creates an Integrity Report (IR) containing both the quote and the ML, and sends it to the challenger. Finally, the challenger verifies the IR, ensuring that the quote is fresh and authentic, the ML has not been tampered with, and the measurements within the ML reflect a trustworthy system.

4.4.2 Keylime Policy

To ensure the integrity of the system, it is necessary to establish a set of rules that define what is considered trusted and compliant. While IMA provides the capability to measure files, binaries, and configurations loaded into a system, it does not, by itself, define how these measurements should be evaluated or what actions should be taken in case of an integrity violation. This is done by Keylime policies. A policy is a structured set of rules that dictate how a system's integrity is assessed and enforced during remote attestation. These policies define the expected state of the system by specifying valid configurations, expected hashes for measured files, and the conditions under which a machine should be considered compromised.

At the core of a policy, there is the definition of reference values against which system measurements are compared. These reference values typically include precomputed hashes of critical system files, expected configurations, and known-good values for TPM PCRs. Since IMA logs

and extends measurements into the PCR 10, Keylime retrieves these values and cross-checks them with the policy's expected state. If a measurement deviates from what is defined in the policy, it indicates a potential integrity violation. Keylime policies also specify how to handle violations and anomalies. When an unexpected measurement is detected, the policy can trigger various responses depending on the severity of the deviation. In general, a runtime policy can contain different rules, such as:

- Whitelist of File Hashes: Specifies acceptable cryptographic hashes for files or executables on the system.
- IMA Rules: Integrates with Linux's IMA to monitor file access and usage.
- Excludelist of files: filename containing a list of paths to be excluded from monitoring.

4.4.3 IMA Template

Each entry in an ML contains information representing a Measurement Entry(ME). An IMA template defines what specific data about the ME should be recorded in the IMA measurement list and displayed in the ML files. To allow ML entries to store several metadata, a template management mechanism was introduced in Linux 3.13.0. This mechanism relies on two key data structures:

- A template field, specifies a type of data stored within an ME.
- A template descriptor, which defines the template fields included in an ME.

PCR	template-hash	template-name	filedata-hash	filename-hint
10	5866yndmdz[...]5210pc57sc	ima-ng	sha1:a9e3c5[...]f4d2b1	/usr/bin/kmod
10	5349yjanpv[...]7691n8zdnp	ima-ng	sha1:73f1b2[...]ed7a6c	/usr/libexec/lsd
10	4666wnoe2c[...]2498g1g1j9	ima-ng	sha1:1a2b8d[...]f7c9e4	/bin/bash
10	2919sthazh[...]3936run4w2	ima-ng	sha1:c3d2e9[...]ab4f7b	/etc/network/interfaces
10	60882idkpv[...]9037njkk1n	ima-ng	sha1:9a4c7f5[...]b2f0a3	/var/cache/apt/archives
...	...	...	...	...

Table 4.1. IMA ML with the ima-ng template

Figure 4.1 illustrates an example of an ML created using the ima-ng template. The fields, listed from left to right, include:

- The PCR index where the entry was extended (usually PCR 10).
- The template hash is computed over the template fields using SHA-1.
- The template name assigned to the entry.
- The event data hash is computed over the file's contents or, in the case of boot_aggregate, over the boot-PCRs contents.
- The event name, typically the file's pathname.

4.4.4 Integrity Validation Mechanism

To assess the trustworthiness of the attesting system, the challenger verifies the measurement of the files recorded in the ML against a predefined whitelist of trusted values. As represented in fig. 4.6, if a file path is missing from the whitelist, it indicates that an unrecognized program has been executed. If the path is found, the challenger checks whether the recorded measurement matches an entry in the whitelist. A match confirms the integrity of the file, while a mismatch suggests either an updated version or potential tampering. In such cases, the system policy dictates the necessary actions, often resulting in the system being marked as untrusted unless effective isolation measures are in place.

The integrity of the system can be tracked over time through repeated Remote Attestation cycles. However, for continuous trust to be ensured, measurements collected in different attestations must belong to the same operational period or epoch. If the system is compromised and then rebooted before the next attestation, it could falsely appear as trusted, as the reboot would mask any previous compromise. To address this issue, a mechanism should be in place to detect epoch transitions, such as leveraging TPM counters that can only increment. The BIOS can increase a TPM counter upon each reboot. By incorporating this TPM-signed value into the attestation data, the challenger can determine whether the system has restarted between two attestations. A change in the counter indicates a reboot, while an unchanged value confirms that the ML provides an unbroken record of the system's history.

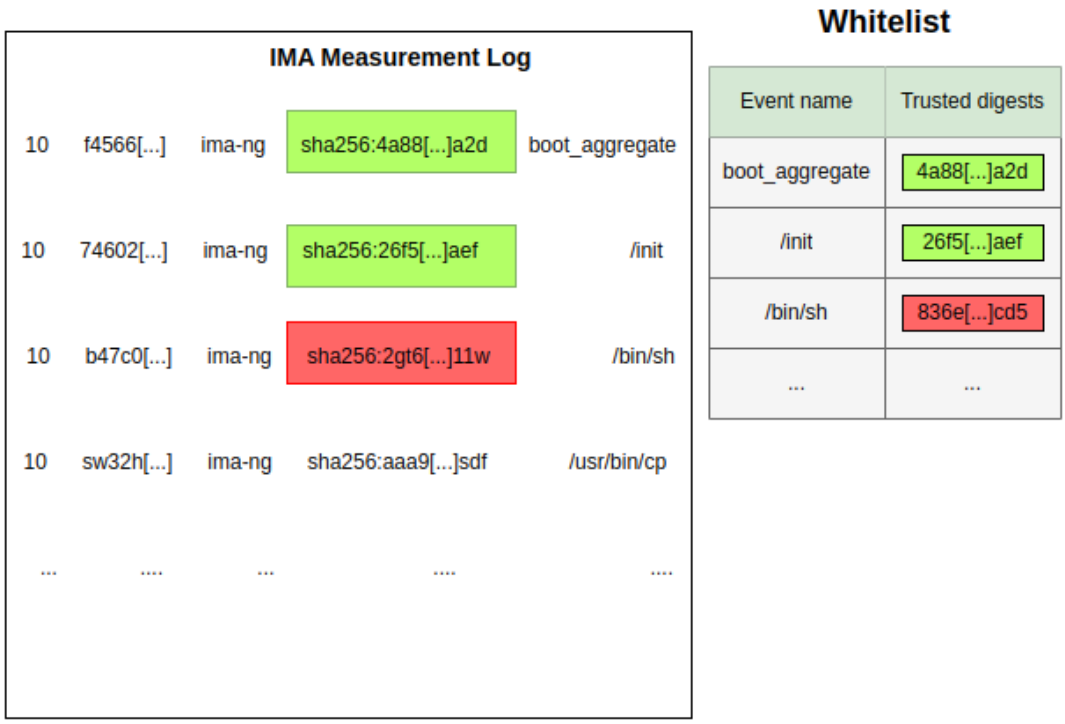

Figure 4.6. Integrity Validation mechanism

4.5 Other functionalities in Keylime

In this section, an overview of the remaining functionalities provided by Keylime is presented. While previous sections covered its core components and roles, this section explores additional features that enhance security, automation, and integration, highlighting Keylime's ability to adapt to various use cases.

- Secure Payloads: secure payloads enable the provisioning of encrypted data to an enrolled node. This encrypted data can be used to deliver secrets needed by the node, such as keys, passwords, certificate roots of trust, etc. Secure payloads are for anything that requires strong confidentiality and integrity to bootstrap the system. The payload is encrypted and sent via the Keylime Tenant CLI (or REST API) to the Keylime Agent. As explained in 4.3.2, the agent also sends part of the key needed to decrypt the payload, a key share. The other key share of the decryption key will be provided to the agent by the Keylime Verifier to decrypt the payload, but only after the agent has met its enrollment criteria. Keylime offers two modes for sending secure payloads: single file encryption and certificate package mode.
  - single file encryption: the user provides a file to the tenant application using the -f option, the tenant encrypts the file using the bootstrap key and securely delivers it to the Agent;
  - package mode: the package mode simplifies various routine actions commonly performed by tenants during Agent provisioning. Firstly, Keylime can automatically set up an X.509 certificate authority and provide native support for certificate revocation.
- User Selected PCR Monitoring: by leveraging the tpm policy feature in Keylime, it is possible to monitor a remote machine for changes in any selected PCR.
- Measured boot: currently, the UEFI firmware has made the event log accessible through an ACPI table, and it is now using this table to expose the boot event log through securityfs. This log is accessible at `/sys/kernel/security/tpm0/binary/bios/measurements`. By combining this feature with secure boot, the designated PCR set can be fully populated, incorporating measurements of all components, up to the kernel and `initrd`. The `initrd`, or initial RAM disk, enables the boot loader to load a RAM disk. This RAM disk can be the primary file system, allowing programs to run from it. Later, a new file system can be mounted from a different device while the original root file system (from `initrd`) is relocated to a directory and can be unmounted as needed. The primary purpose of `initrd` is to facilitate a two-phase system startup. Initially, the kernel initializes with a basic set of built-in drivers. Subsequently, additional modules are loaded from `initrd`. In addition to the boot log data sources mentioned above, users can utilize tpm2-tools to consume the contents of such logs and reconstruct the contents of PCRs [0-9] (and potentially PCRs [11-14]).

  Keylime can leverage this newfound capability with great flexibility. It allows the Keylime operator to specify a "measured boot reference state" or mb_refstate for brevity. This operator-provided data is then utilized by the keylime verifier in a manner akin to the "IMA policy". The keylime verifier compares the information received from the keylime agent against this reference state to ensure integrity and security.

Chapter 5

Quantum-safe Remote Attestation in Keylime

5.1 Idea

The chapter describes the proposed implementation work to enhance the security of the remote attestation process in Keylime by integrating post-quantum cryptography. Since no physical TPM currently supports post-quantum cryptography, direct integration at the hardware level is not yet feasible. Given this limitation, the idea is to extend the attestation Agent and the Verifier to integrate post-quantum cryptography at the software level. This solution allows experimentation with different post-quantum signature schemes while maintaining compatibility with current TPM implementations, ensuring a smooth transition once hardware-based post-quantum TPMs become available.

The idea is described as follows: the Verifier requests proof of integrity from the attester. In response, the attester queries the TPM, which generates a quote (a signed statement of the state of the system) using its conventional cryptographic key (the attestation identity key). Before transmitting the quote, the attester further signs the quote with a new post-quantum key, adding an extra layer of security. Once the Verifier receives the attestation data, it validates the system's integrity by checking both the TPM-generated quote and the post-quantum signature. Specifically, four signature schemes have been selected to perform the signature of the quote:

- **SPHINCS-SHAKE256-SIMPLE and SPHINCS-SHAKE256-FAST**: SPHINCS+ SIMPLE prioritizes security with more hashing, making it slower but more robust. SPHINCS+ FAST optimizes speed by reducing computations.
- **DILITHIUM_5**: DILITHIUM offers a good balance between performance and efficiency, with a very high level of security compared to other post-quantum schemes.
- **MAYO_5**: an algorithm designed to provide compact signatures and fast operations, which makes it particularly suitable for resource-constrained environments.

5.2 Configuration of runtime integrity monitoring

This section provides a detailed explanation of how runtime integrity monitoring operates in the standard Keylime implementation, focusing first on the remote attestation process before adding any modifications introduced in this thesis. It will serve as the foundation for the following section, which will outline the enhancements and additions implemented.

5.2.1 Node Registration Protocol

This phase establishes the initial communication and trust between the components involved in the remote attestation workflow, specifically the registrar and the agent. To execute this protocol, two services must be started, each in its terminal window: one for the registrar and one for the agent. To start the registrar service, open a terminal and execute the following command:

```
$ keylime_registrar
```

This command launches the registrar process, which listens for registration requests from agents on its designated network port. To start the agent service, open a second terminal and execute the following command:

```
$ RUST_LOG=keylime_agent=trace cargo run --bin keylime_agent
```

This command starts the agent service in debug mode and generates detailed log output. This verbosity is useful for troubleshooting and understanding the agent's behavior during the registration process. Once both services are running, the registration protocol proceeds as follows.

- The agent sends a registration request to the registrar, including the endorsement key, the attestation identity key, the IP address, the port, the certificates, and other registration parameters.
- The registrar validates the agent's credentials and securely stores its keys.
- Upon successful completion of this exchange, the agent is officially registered and ready to participate in the subsequent phases of the remote attestation process.

5.2.2 Starting the Keylime Verifier

After successfully starting the registrar and agent services, the next steps involve launching the verifier. To start the verifier service, open a terminal and execute the following command:

```
$ keylime_verifier
```

This command initializes the verifier, which listens for requests from tenants and orchestrates the verification of attestation data provided by the agent. The verifier maintains a database of registered agents and their associated trust policies. It ensures the verifier is running and properly connected to the registrar before proceeding to the next steps.

5.2.3 Initiating the Remote Attestation Process

The tenant is the component responsible for initiating and managing the remote attestation process for a specific agent. To start the tenant to begin the attestation, execute the following command:

```
$ keylime_tenant -c add --uuid d432fbb3-d2f1-4a97-9ef7-75bd81c00000
    --runtime-policy /path/to/policy.json
```

Where:

- "-c" specifies the action to be performed.
- "–uuid" specifies the unique identifier (UUID) of the agent added. The UUID uniquely identifies the agent in the Keylime infrastructure. The example shows the default UUID used by Keylime.
- "–runtime-policy" specifies the path to a JSON file containing the runtime policy for the agent.

Once the tenant command is executed:

- The tenant communicates with the registrar to retrieve the agent's public keys and verify its identity.
- The tenant shares the runtime policy with the verifier and requests the attestation of the agent.
- The verifier periodically contacts the agent, retrieves its integrity measurements, and compares them to the runtime policy.
- If the measurements match the expected values in the policy, the agent is considered trustworthy. Otherwise, the verifier flags a policy violation, stopping the remote attestation.

5.3 Creation of the runtime policy

A runtime policy is a collection of "golden" cryptographic hashes that represent the untampered state of files or the keys authorized to be loaded into keyrings for IMA verification. The runtime policy is uploaded to the Keylime Verifier, which then regularly polls TPM quotes for PCR 10 on the agent's TPM. It compares the current state of the agent's files to the expected hashes defined in the policy. If the object has been tampered with or an unexpected key was loaded onto a keyring, the hashes will not match and Keylime will place the agent into a failed state.

5.3.1 Configuration of IMA

Keylime's runtime integrity monitoring requires the setup of Linux IMA. It is needed to deploy an ima policy file. This file should be located in $/etc/ima/ima-policy$. Next, to configure the IMA policy, create the file $/etc/ima/ima-policy$ and populate it with the following script:

```
# PROC_SUPER_MAGIC
dont_measure fsmagic=0x9fa0
# SYSFS_MAGIC
dont_measure fsmagic=0x62656572
# DEBUGFS_MAGIC
dont_measure fsmagic=0x64626720
# TMPFS_MAGIC
dont_measure fsmagic=0x01021994
# RAMFS_MAGIC
dont_measure fsmagic=0x858458f6
# SECURITYFS_MAGIC
dont_measure fsmagic=0x73636673
# SELINUX_MAGIC
dont_measure fsmagic=0xf97cff8c
# CGROUP_SUPER_MAGIC
dont_measure fsmagic=0x27e0eb

measure func=BPRM_CHECK mask=MAY_EXEC
measure func=FILE_MMAP mask=MAY_EXEC
```

This default policy measures all executables and all files mmapped as executable in file_mmap and module checks and skips several irrelevant files (logs, audit, tmp, etc). At this point, go to file /etc/default/grub and set the value

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash ima_policy=tcb"
```

this line specifies kernel parameters passed to the Linux kernel at boot time via the GRUB bootloader. In particular:

- "quiet" option suppresses most of the kernel's boot messages during startup. Instead of displaying detailed logs, the boot process will show minimal or no text on the screen. This is often used to make the boot process look cleaner.
- "splash" option enables the display of a graphical splash screen during boot, replacing the textual output of the kernel messages.
- "ima_policy=tcb" is a kernel parameter related to the IMA. It sets the IMA policy to Trusted Computing Base (TCB). Under this policy, the IMA subsystem measures and appraises the integrity of files accessed by the kernel that are critical to the system's security, such as executables, shared libraries, and kernel modules. This feature is often used in environments where system integrity and security are critical, such as servers or systems adhering to strict security policies.

After modifying the grub file, the changes must be applied by running the command:

```
sudo update-grub
sudo reboot
```

This regenerates the GRUB configuration file (`/boot/grub/grub.cfg`) to include the updated kernel parameters. It is possible to verify IMA is measuring your system by checking the content of this file:

```
sudo nano /sys/kernel/security/integrity/ima/ascii_runtime_measurements
```

Then, create a basic policy by using an IMA measurement log from the system:

```
keylime_create_policy -m /path/to/ascii_runtime_measurements -o
    runtime_policy.json
```

5.4 Integration of Liboqs

This section focuses on liboqs, an open-source library developed by the Open Quantum Safe (OQS) project to support quantum-resistant cryptographic algorithms. In this work, liboqs was utilized to implement the signature process for the quote generated during Keylime's remote attestation process. Specifically, the algorithms `SPHINCS+-SHAKE-256s-simple`, `SPHINCS+-SHAKE-256f-simple`, `Dilithium5`, and `MAYO-5` have been used. The implementation involved designing and developing the key primitives of a digital signature process:

1. Key Generation: the process of creating a private and public key pair. The private key is used to sign messages, while the public key is shared to verify those signatures.
2. Signature Generation: using the private key to compute the signature for a given message M. This ensures the authenticity and integrity of the message.
3. Signature Verification: validating the signature's authenticity and ensuring the message M has not been tampered with, using the associated public key.

5.4.1 Creation of the keypair

The function `generate_PQ_keypair` is implemented to generate a pair of cryptographic keys using the algorithms provided by the liboqs library.

- Initialization of the Keypair Result Structure: a custom data structure called KeypairResult is defined to store the generated keys and their respective lengths.

- Creating the Signature Object: a new post-quantum signature object is created using the `OQS_SIG_new` function, which initializes the internal structures required for the key generation process. If the signature object cannot be created, an error is reported, and the function terminates.
- Keypair Generation: with the allocated memory in place, the actual keypair is generated using the `OQS_SIG_keypair` function. This step represents the core operation of the function and uses quantum-safe algorithms to produce secure keys.

5.4.2 Generation of the signature

The `sign_with_PQ` function generates a digital signature for the quote. The function returns a `SignatureResult` structure containing the generated signature.

```
typedef struct {
    uint8_t* signature;
    size_t signature_len;
} SignatureResult;
```

This structure is initialized with default values (NULL for the signature pointer and zero for its length). The first operational step is to verify whether the post-quantum algorithm is available and supported by the liboqs library. Once the algorithm's availability is confirmed, the function initializes a post-quantum signature object using the `OQS_SIG_new` function. The signing operation is performed using the `OQS_SIG_sign` function:

```
if (OQS_SIG_sign(sig, result.signature, &sig->length_signature, quote,
    quote_len, pq_priv_key) != OQS_SUCCESS) {
   fprintf(stderr, "Error in the signature!\n");
   free(result.signature);
   OQS_SIG_free(sig);
   return result;
}
```

5.4.3 Verification of the signature

The `verify_signature` function is responsible for verifying the validity of the digital signature generated. The core of the function is the signature verification process, performed using the `OQS_SIG_verify` function.

```
if (OQS_SIG_verify(sig, message, quote_len, signature,
    sig->length_signature, public_key)!= OQS_SUCCESS) {
   return -1;
}
```

5.5 Changes in Keylime

This section outlines the modifications made to the Keylime code to integrate support for the post-quantum signature in the remote attestation process. The implementation requires modifications to the Registrar, the Agent, the Verifier, and the Tenant.

5.5.1 Agent

The first change in the Keylime agent involves modifying the registration process. This phase is critical as it establishes the initial communication between the agent and the registrar, where the agent sends essential information to the registrar so it can be stored in the registrar's database

for future interactions. This process has been described in detail in section 4.3.1. In this implementation, the agent generates a post-quantum keypair using the chosen algorithm and includes the public key in the data sent to the registrar during registration. This additional step ensures the agent's identity and integrity can be verified using quantum-safe mechanisms.

```
let pq_result = unsafe {
      generate_PQ_keypair(1)
  };
```

Listing 5.1. Call to the function

The `generate\_PQ\_keypair` function returns a structure containing the two generated keys: the public and the private key.

```
#[link(name = "gen_keypair")]
    extern "C" {
        fn generate_PQ_keypair(stampa: i32) -> KeypairResult;
    }

#[repr(C)]
    struct KeypairResult {
        public_key: *const u8,
        public_key_len: size_t,
        private_key: *const u8,
        private_key_len: size_t,
    }
```

To facilitate storage and transmission, the public key generated by the agent is encoded in Base64 format before being included in the registration payload.

```
 let public_key_vec = unsafe {
      if !pq_result.public_key.is_null() {
          Vec::from(std::slice::from_raw_parts(
              pq_result.public_key,
              pq_result.public_key_len,
          ))
      } else {
          Vec::new()
      }
  };
let pq_key_base64: String =
    general_purpose::STANDARD.encode(&public_key_vec);
```

Listing 5.2. Base64 conversion

Encoding the key in Base64 offers several advantages: it ensures compatibility with text-based protocols, avoids issues related to binary data handling, and simplifies integration with databases and logging systems. This encoded key is then securely transmitted to the registrar. The enhanced registration payload, now including the Base64-encoded post-quantum public key, is transmitted securely to the registrar. The registrar is then responsible for storing this key in its database along with the other details provided by the agent.

```
let data = Register {
   ekcert,
   ek_tpm,
   aik_tpm,
   iak_tpm,
   idevid_tpm,
   idevid_cert,
   iak_cert,
   iak_attest,
```

```
        iak_sign,
        mtls_cert,
        ip,
        port: Some(port),
        pq_key: pq_key.clone(),
    };
```

Listing 5.3. Structure of the registration payload

In particular, if we focus just on the keys, fig. 5.1 shows the updated node registration workflow, where the cloud agent sends to the registrar:

- its UUID.
- the public Attestation Identity Key: the Attestation Identity Key is an asymmetric key pair generated inside the TPM. It allows the TPM to prove its integrity without exposing the EK. The AIK signs the integrity reports generated by the TPM. These reports demonstrate the current state of the system, including measurements of the operating system and software.
- the public Endorsement Key: it is a unique, asymmetric key pair generated and embedded in a TPM during its manufacturing process. It is used to establish trust in the TPM itself. The EK is used to prove the authenticity of the TPM to external entities. This ensures that the TPM is legitimate and has not been tampered with.
- the public Post-Quantum Key.

Then, the registrar generates a challenge using an ephemeral key, and if the Agent correctly solves the challenge, it sends back the response signed with the private post-quantum key. The registrar verifies the signature using the post-quantum public key, and if the response is correct, the keys are stored in the registrar's DB. After the registration process, in the remote attestation phase, the agent generates and sends a `PQquote` (Post-Quantum Quote) to the verifier. This `PQquote` represents an enhancement of the standard Keylime quote, incorporating additional fields required for post-quantum security. Specifically, it includes:

- The post-quantum signature
- The length of the signature
- The post-quantum public key
- The length of the public key
- The hash algorithm used during the generation of the post-quantum signature

```
    let pq_quote = PQquote {
        sign_pq: signature_vec.clone(),
        sign_pq_len: result.signature_len,
        pq_key: data.pq_pub_key.clone(),
        pq_key_len: data.pq_pub_key_len,
        hash_alg_pq: "shake_256".to_string(),
        quote_len: quote.quote.len(),
        quote: quote.quote,
        hash_alg: quote.hash_alg,
        enc_alg: quote.enc_alg,
        sign_alg: quote.sign_alg,
        pubkey: quote.pubkey,
        ima_measurement_list: quote.ima_measurement_list,
        mb_measurement_list: quote.mb_measurement_list,
        ima_measurement_list_entry: quote.ima_measurement_list_entry,
    };
```

Listing 5.4. Structure of the Post-quantum quote

The generation of the post-quantum signature is performed by invoking a dedicated function. This function handles the task of signing the data using the post-quantum algorithm. The integration of this functionality within the agent relies on smooth interaction between the Rust and C source bases, ensuring efficient execution of the signing process while maintaining system compatibility.

```
 let result = unsafe {
    sign_with_PQ(quote_ptr,
        quote.quote.len(),pq_priv_key_cstring,data.pq_priv_key_len)
};
```

Once the PQquote is fully constructed, it is transmitted to the verifier.

```
let response = JsonWrapper::success(pq_quote);
HttpResponse::Ok().json(response)
```

5.5.2 Registrar

From the registrar's perspective, the process begins when the agent sends its public keys to the registrar. These keys include the post-quantum public key. The registrar in the registration phase, receives the response of the agent signed with the private post-quantum key, as shown in fig. 5.1. The correctness of this key is checked by verifying the signature with the public post-quantum key sent in the first part of the registration by the agent. If the Verification of the signature is performed correctly, the registrar has to store the public key. To accommodate this new key, it is necessary to extend the registrar's database schema by adding a new field that can store the post-quantum public key.

The modified database structure, shown below, includes the new field `pq_key`, which is of type $String(500)$:

```
class RegistrarMain(Base):
__tablename__ = "registrarmain"
agent_id = Column(String(80), primary_key=True)
key = Column(String(45))
aik_tpm = Column(String(500))
ekcert = Column(String(2048))
ek_tpm = Column(String(500))
iak_tpm = Column(String(500))
iak_cert = Column(String(2048))
...
pq_key = Column(String(500))
```

After adding this new field to the database schema, modifications are also required in the registrar's logic for processing the agent's registration. Specifically, the registrar must extract the `pq_key` from the agent's registration payload and store it in the database. This involves extending the registrar's GET request logic to include the post-quantum public key in the received data.

The following line is added to the registrar's data extraction logic:

```
"pq_key": agent.pq_key,
```

Once the public key is extracted, it is stored in the database by extending the dictionary of values used for the database insertion. To achieve this, the POST request logic must be modified. The updated dictionary is shown below:

```
d: Dict[str, Any] = {
    "agent_id": agent_id,
    "ek_tpm": ek_tpm,
    "aik_tpm": aik_tpm,
    "ekcert": ekcert,
    "iak_tpm": iak_tpm,
```

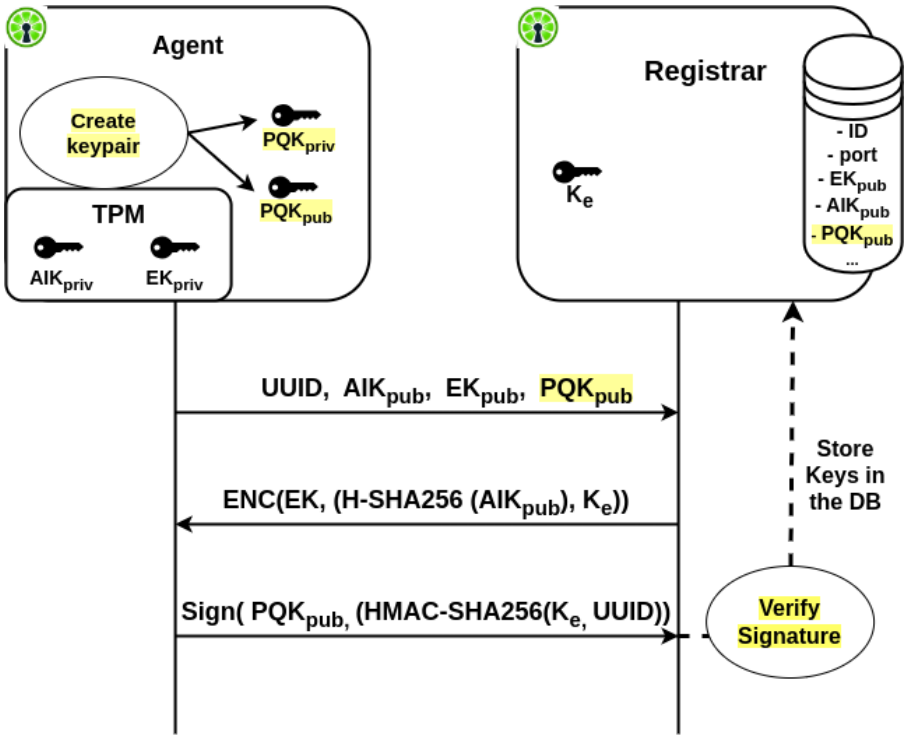

Figure 5.1. Registration of the agent

```
        "idevid_tpm": idevid_tpm,
        "iak_cert": iak_cert,
        "idevid_cert": idevid_cert,
        "ip": contact_ip,
        "mtls_cert": mtls_cert,
        "port": contact_port,
        "virtual": int(ekcert == "virtual"),
        "active": int(False),
        "key": key,
        "provider_keys": {},
        "regcount": regcount,
        "pq_key": pq_key,
    }
```

Listing 5.5. Insertion of values in the DB

To enhance the robustness and security of the RegistrarAgent class, a validation function is introduced to validate the integrity of the `pq_key` field.

```
def _validate_pq_key(self, pq_key: str) -> None:
   if not isinstance(pq_key, str):
      raise ValueError("pq_key must be a string")
   try:
     decoded_key = base64.b64decode(pq_key)
   except Exception:
     raise ValueError("Invalid pq_key: not a valid Base64 string")
```

Listing 5.6. Validation function

The function is integrated directly into the update method to ensure all modifications to the `pq_key` undergo these checks:

```
def update(self, data):
```

```
self.cast_changes(
    data,
    ["agent_id", "ek_tpm", "ekcert", "aik_tpm", "iak_tpm", "iak_cert",
        "idevid_tpm", "idevid_cert", "ip","pq_key"]
    + ["port", "mtls_cert"],
)

# Verify EK as valid
self._check_ek()
# Verify IAK/IDevID as valid and trusted
self._check_iak_idevid(data.get("iak_attest"), data.get("iak_sign"))
# Ensure either an EK or IAK/IDevID is present, depending on
    configuration
self._check_root_identity_presence()
# Handle certificates which are not fully compliant with ASN.1 DER
self._check_all_cert_compliance(data)

...

# Basic validation of pq_key
self._validate_pq_key(data.get("pq_key"))
```

5.5.3 Verifier

This section focuses on the modifications made to the verifier component of Keylime; specifically, the changes were applied to the `invoke_get_quote` function. This function is responsible for receiving the integrity quote from the Keylime agent through an HTTP GET request and verifying its correctness by invoking another function, `process_quote_response`.

The new implementation extracts additional fields required for verifying the post-quantum signature from the HTTP response sent by the agent. These fields include the post-quantum signature, the corresponding public key, and their respective lengths. The extraction of these fields was implemented as follows:

```
quote = json_response.get("results", {}).get("quote").encode('utf-8')
quote_len = json_response.get("results", {}).get("quote_len")
sign_pq = bytearray(json_response.get("results", {}).get("sign_pq"))
sign_pq_len = json_response.get("results", {}).get("sign_pq_len")
pq_key = bytearray(json_response.get("results", {}).get("pq_key"))
pq_key_len = json_response.get("results", {}).get("pq_key_len")
```

Here:

- quote: The integrity quote sent by the agent, extracted as a UTF-8 encoded byte string.
- quote_len: The length of the quote field.
- sign_pq: The Post-Quantum signature of the quote, extracted as a byte array.
- sign_pq_len: The length of the Post-Quantum signature.
- pq_key: The public key corresponding to the Post-Quantum signature, extracted as a byte array.
- pq_key_len: The length of the public key.

The verifier retrieves the Post-Quantum public key associated with the agent from the registrar, but the Post-Quantum public key is not saved in its database to reduce the risk of exposure. Instead, the key is temporarily stored in the `exclude_db` dictionary. The `exclude_db`values are excluded from the data persisted in the database, ensuring that the public key remains ephemeral.

```
exclude_db["pq_key"] = json_body["pq_key"]

# Retrieve PQ public key from registrar
pq_key_registrar = bytearray(base64.b64decode(exclude_db["pq_key"]))

# Compare the registrar's key with the agent's key
if pq_key == pq_key_registrar:

                logger.info("PQ key received correctly \n")
                result = lib.verify_signature(quote_ptr, c_size_t(quote_len),
                    signature_ptr, c_size_t(sign_pq_len), public_key_ptr)
                ....
else:
                failure = Failure(Component.QUOTE_VALIDATION)
                failure.add_event("invalid post-qaunutm signature",{"message":
                    "post-qauntum Public Key is not corresponding to the
                    correct one"},False)
                ....
```

A function called `verify_signature` verifies the post-quantum signature. The verification result is returned as a Boolean value (`result`) indicating whether the verification is successful.

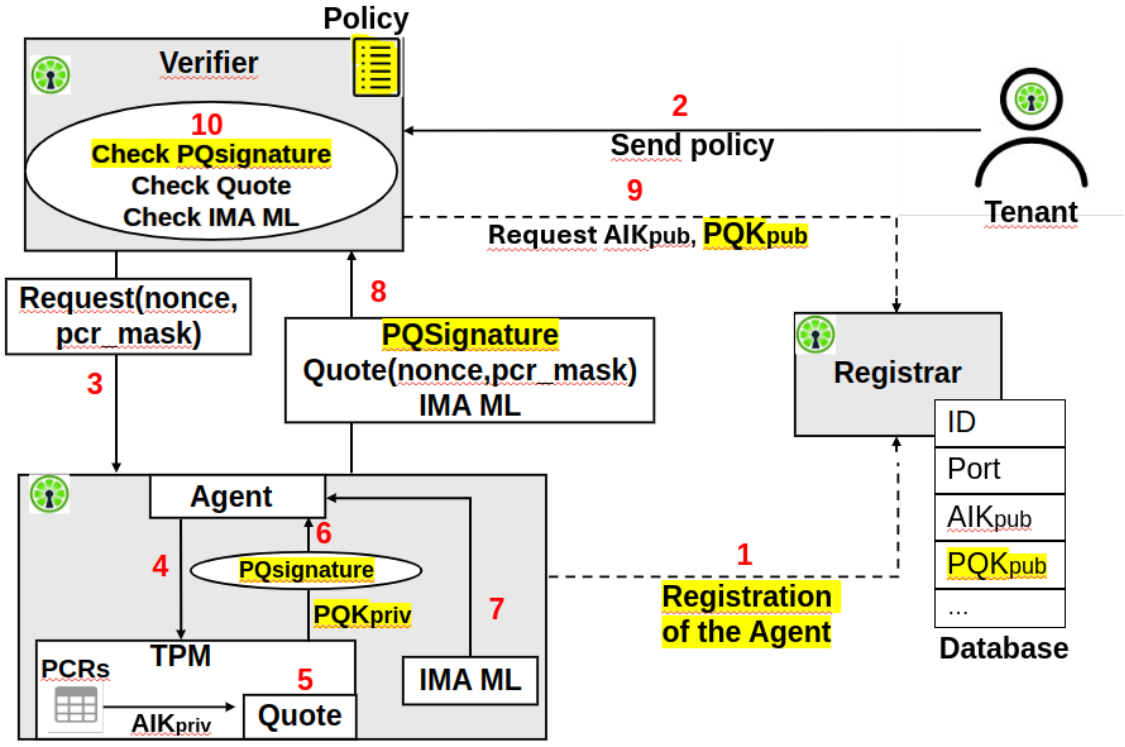

Figure 5.2. Remote attestation with post-quantum integration

The final result is shown in fig. 5.2: the first phase is the registration phase, where the Agent stores his information, including the post-quantum public key, into the registrar DB; then, the Tenant sends the policy/allowlist to the Verifier, created following the rules indicated in the "ima-policy" file. At this point, the Verifier sends the request to the agent, including the nonce and the pcr_mask. The Agent sends to the TPM the request of the generation of the quote, so the TPM signs with the private attestation identity key the PCR values and sends back the quote to the Agent. The Agent signs the quote with the post-quantum private key, obtaining the post-quantum signature. Finally, he retrieves the measurements collected by IMA and sends to the Verifier the

response, containing the post-quantum signature, the quote, and the IMA Measurement List. The Verifier retrieves from the Registrar the needed keys (the post-quantum public key and the public attestation identity key), and checks the correctness of the information received. If everything is proved to be correct, the Verifier sends a new request of the quote, and the cycle restarts.

5.5.4 Updating the Registrar Table to add the Post-Quantum key field

To modify the registrarmain table in the SQLite database to include the `pq_key` field, several steps must be performed.

Renaming the Existing Table

The original registrarmain table was renamed to `registrarmain_old` to preserve its structure and data during the migration process:

```
ALTER TABLE registrarmain RENAME TO registrarmain_old;
```

Creating the Updated Table

A new table named registrarmain was created with the updated schema. The `pq_key` field is defined as a BLOB type to properly store its binary data:

```
CREATE TABLE registrarmain (
 agent_id VARCHAR(80) PRIMARY KEY,
 key VARCHAR(45),
 ekcert TEXT,
 virtual BOOLEAN,
 active BOOLEAN,
 provider_keys TEXT,
 regcount INTEGER,
 aik_tpm VARCHAR(500),
 ek_tpm VARCHAR(500),
 ip VARCHAR(15),
 port INTEGER,
 mtls_cert TEXT,
 iak_tpm VARCHAR(500),
 idevid_tpm VARCHAR(500),
 iak_cert VARCHAR(2048),
 idevid_cert VARCHAR(2048),
 pq_key BLOB
);
```

Migrating Data from the Old Table

Data from `registrarmain_old` are transferred to the new registrarmain table. This ensures that all existing data, along with the `pq_key` field, are preserved during the migration:

```
 INSERT INTO registrarmain (
 agent_id, key, ekcert, virtual, active, provider_keys, regcount,
 aik_tpm, ek_tpm, ip, port, mtls_cert, iak_tpm, idevid_tpm, iak_cert,
 idevid_cert, pq_key
)
SELECT
 agent_id, key, ekcert, virtual, active, provider_keys, regcount,
 aik_tpm, ek_tpm, ip, port, mtls_cert, iak_tpm, idevid_tpm, iak_cert,
 idevid_cert, pq_key
FROM registrarmain_old;
```

Dropping the Old Table

Once the data migration is confirmed to be successful, the old table is removed to free up space and avoid redundancy:

```
DROP TABLE registrarmain_old;
```

Validating the Updated Table

To verify the new structure of the registrarmain table, the following SQLite command is used:

```
PRAGMA table_info(registrarmain);
```

```
sqlite> PRAGMA table_info(Registrarmain);
0|agent_id|VARCHAR(80)|0||1
1|key|VARCHAR(45)|0||0
2|ekcert|TEXT|0||0
3|virtual|BOOLEAN|0||0
4|active|BOOLEAN|0||0
5|provider_keys|TEXT|0||0
6|regcount|INTEGER|0||0
7|aik_tpm|VARCHAR(500)|0||0
8|ek_tpm|VARCHAR(500)|0||0
9|ip|VARCHAR(15)|0||0
10|port|INTEGER|0||0
11|mtls_cert|TEXT|0||0
12|iak_tpm|VARCHAR(500)|0||0
13|idevid_tpm|VARCHAR(500)|0||0
14|iak_cert|VARCHAR(2048)|0||0
15|idevid_cert|VARCHAR(2048)|0||0
16|pq_key|VARCHAR(500)|0||0
```

Figure 5.3. Registrar Database

The output of the command is shown in fig. 5.3.

Viewing and Validating Data in the Database

To inspect the updated table and confirm the presence and integrity of the `pq_key` field, the following commands are executed:

Open the SQLite database:

```
sqlite3 /var/lib/keylime/reg_data.sqlite
```

Query the registrarmain table to view specific fields, such as `agent_id` and `pq_key`:

```
SELECT agent\_id, pq_key FROM registrarmain;
```

By following these steps, the database schema is successfully updated to include the `pq_key` field while preserving all existing data.

5.6 Remote Attestation Failure

To test a failure scenario in Keylime Remote Attestation, it is necessary to create a script with arbitrary content (e.g., echo "hello world") that is not part of the pre-defined runtime policy. Then, execute the script as the root user on the agent machine. The Verifier's output will report the agent's status change to "failed".

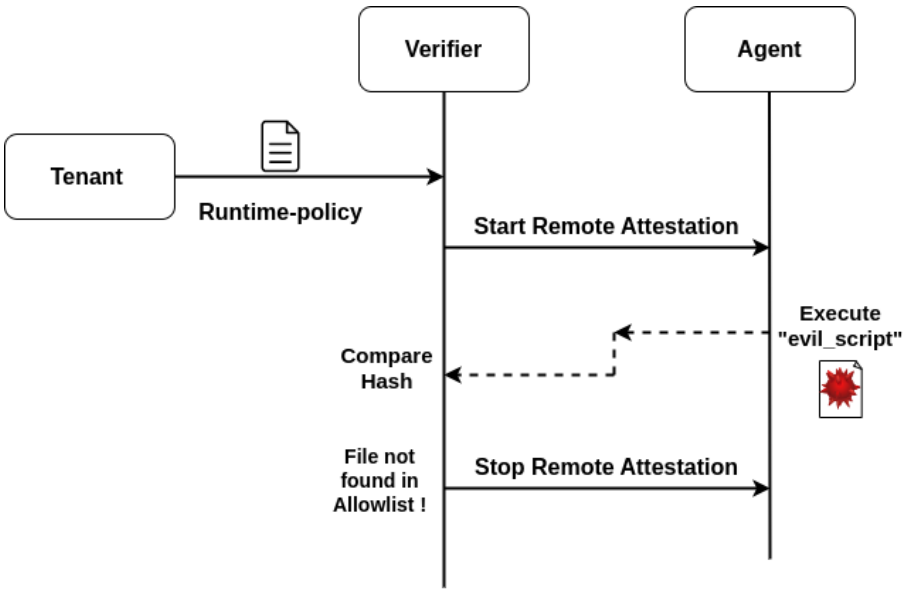

Figure 5.4. Attestation failure

What happens is that the Agent continuously measures the hash values of monitored files using the Linux IMA. These hash values are sent to the Verifier for comparison against the allowlist provided in the Runtime Policy. So when the Agent executes a script (evil_script.sh) that is not part of the allowlist, the Verifier detects a mismatch between the reported hash value and the allowlist, updates the Agent's attestation status to "failed" and stops the remote attestation. There will be the following output on the verifier showing the agent status change to failed:

```
keylime.tpm - INFO - Checking IMA measurement list...
keylime.ima - WARNING - File not found in allowlist: /root/evil_script.sh
keylime.ima - ERROR - IMA ERRORS: template-hash 0 fnf 1 hash 0 good 781
keylime.cloudverifier - WARNING - agent
    D432FBB3-D2F1-4A97-9EF7-75BD81C00000 failed, stopping polling
```

Chapter 6

Testing

This chapter presents the results of the tests conducted on the Keylime framework, which has been modified to support quantum-safe remote attestation. The evaluation includes functional tests to validate the correct operation of the attestation process and performance tests to measure latency and resource consumption during the attestation process.

6.1 Testbed

The assessment of the operational efficiency and effectiveness of the suggested solutions involves executing tests on a system equipped with an Intel i5-1035G1 processor clocked at 1.00 GHz and capable of reaching a maximum frequency of 3.60 GHz, featuring 4 cores, 8 threads, 16 GB of RAM, and a TPM2.0 chip. This system runs on Ubuntu 24.04 LTS and operates with a Linux kernel version 6.8.0. The environment includes Liboqs version 0.11.0 and a customized version of Keylime 7.11.0.

The attester and Verifier are on two different machines:

- One machine hosts the Keylime Tenant, Registrar, and Verifier components.
- Another machine runs the Keylime Agent.

For instructions on compiling and installing the testbed configuration, please refer to Appendix A. You can find steps for installing and configuring Keylime in Appendix A.2 and instructions for installing Liboqs in Appendix A.5.

6.2 Functional tests

The purpose of functional tests is to verify whether the software implementation of the proposed solution meets the requirements. Specifically, these tests check that the Agent's registration with the Registrar and the remote attestation process function correctly. For these tests, the `SPHINCS+-SHAKE256-SIMPLE` variant is used for signature generation and verification. Later, in performance tests, comparisons will be made between the `SIMPLE` and the `FAST` versions of `SPHINCS+`, and with `Dilithium_5` and `Mayo_5`.

6.2.1 Tests of Agent registration

The first test focuses on the Agent's registration process with the Registrar. This involves ensuring that the Agent can successfully register and that the SPHINCS public key is correctly transferred to the Registrar. To perform this test an Agent initiates the registration process.

Additionally, the time required to complete the registration is measured to ensure it falls within an acceptable threshold, which should be defined based on the system's expected use cases. To test the Agent registration process, two terminals are used: one to start the Registrar and the other to run the Agent. The Registrar is started using the `keylime_registrar` command. The process logs several important steps:

```
INFO - Starting Keylime registrar...
INFO - Reading configuration from ['/etc/keylime/registrar.conf']
INFO - database_url is set, using it to establish database connection
```

The Registrar starts by reading the configuration file (registrar.conf), which contains settings such as database configurations, network parameters, etc. Then, it establishes a connection to the database. This message confirms that it is configured to use a database (`SQLite` in this case) to store Agent information and other relevant data.

```
POST
   /v2.2/Agents/d432fbb3-d2f1-4a97-9ef7-75bd81c00000
INFO - EK received for Agent 'd432fbb3-d2f1-4a97-9ef7-75bd81c00000'
INFO - Sphincs public key registered correctly

Begin PQ Public KEY (Base64 encoded)-----
       +mQUlXZKu7J4oMbj5x0ctse8Zshlvrm21ae1ldMGx3U3
       +VrS5S4INC7/EPuyqjxERiV2GT11VzrypH3hM4GLYA==
-----End PQ Public KEY
```

The Registrar receives a POST request to register an Agent. The request is sent to the /v2.2/Agents/UUID endpoint, where `d432fbb3-d2f1-4a97-9ef7-75bd81c00000` is the unique identifier `(UUID)` of the Agent. The Registrar successfully receives the Endorsement Key from the Agent. The Registrar logs the `base64-encoded` post-quantum public key that it has received from the Agent.

```
INFO - Sent 200 in 32ms
INFO - PUT
         /v2.2/Agents/d432fbb3-d2f1-4a97-9ef7-75bd81c00000
INFO - Sent 200 in 31ms
```

The Registrar sends a `HTTP response` with status code `200 OK` to indicate that the Agent's registration is successful. It took 32 milliseconds to process the request. The Registrar then receives a `PUT request` to update the Agent's information. This likely indicates that the Agent is now being activated. On the Agent side:

```
keylime_Agent::registrar_agent > Send PQ public key to the registrar

Begin PQ Public KEY (Base64 encoded)-----
+mQUlXZKu7J4oMbj5x0ctse8Zshlvrm21ae1ldMGx3U3
+VrS5S4INC7/EPuyqjxERiV2GT11VzrypH3hM4GLYA==
-----End PQ Public KEY

keylime_agent > SUCCESS: Agent d432fbb3-d2f1-4a97-9ef7-75bd81c00000
    registered
keylime_agent > SUCCESS: Agent d432fbb3-d2f1-4a97-9ef7-75bd81c00000
    activated
```

The Agent sends its post-quantum public key encoded in base64 format and then logs that it has successfully registered and activated with the Registrar. The logs from both the Registrar and the Agent provide valuable information about the time taken for the registration process. By analyzing the timestamps, it is possible to calculate the total duration for Agent registration and activation:

```
Registration Step (POST): 32 ms
Activation Step (PUT): 31 ms
Total Time (POST + PUT):
$32 + 31 = 63ms$
```

The total time required for the Agent registration and activation process is approximately 63 milliseconds, demonstrating the system's high efficiency.

6.2.2 Tests of periodic attestation

The second test evaluates the remote attestation process, ensuring the Agent and the Verifier work correctly. A complete attestation session involves the Agent generating a quote, signing it using SPHINCS+, and then verifying the signature on the Verifier side. The total time required for the entire attestation cycle is calculated and analyzed. The Verifier is started using the command `keylime_verifier`. It first registers the Agent successfully, as indicated by the message:

```
POST
   returning 200 response for adding Agent id:
       d432fbb3-d2f1-4a97-9ef7-75bd81c00000
```

Then, the Verifier sends an integrity quote request to the Agent along with the provided nonce. The Agent responds with the requested quote.

```
INFO - Request of Integrity Quote, nonce = 9frP8sK4qEDEgE79tl2V
INFO - Integrity Quote received
```

The Verifier receives and successfully verifies the post-quantum public key and SPHINCS+ signature. Following this, the Verifier proceeds to check the IMA measurement list of the Agent to ensure that no unauthorized changes have occurred in its system.

```
   INFO - PQ key received correctly
Public key: 7476014B68234E68C55D94C7F1ED163E0246195CDE0EF40B1D47A8D964832609
          7FB00768EEA0DA42E5D3D62C3DA3B19C397E2BEDFF5E180B7E785854FB071A50

   INFO - Verification of Sphincs signature: Valid
   INFO - Checking IMA measurement list on Agent:
       d432fbb3-d2f1-4a97-9ef7-75bd81c00000
```

From the Agent's perspective, the process involves handling a series of requests, starting with responding to an Identity Quote request:

```
GET
   invoked from "127.0.0.1"
   with uri /v2.2/quotes/identity?nonce=jadq44bIgEpz2R3Jdp5y
Calling Identity Quote with nonce: jadq44bIgEpz2R3Jdp5y
GET
   identity quote returning 200 response
```

This request specifies the nonce and the IP address of the Verifier server. After successfully sending the identity quote, the Agent receives a request for an integrity quote from the Verifier:

```
GET
   invoked from "127.0.0.1" with uri
       /v2.2/quotes/integrity?nonce=9frP8sK4qEDEgE79tl2V&mask=0x400&partial=0&ima_ml_en
Calling Integrity Quote with nonce: 9frP8sK4qEDEgE79tl2V, mask: 0x400
GET
   integrity quote returning 200 response
```

With a specific nonce and a designated mask, which refers to the PCRs to be used. The Agent processes the integrity quote request with the specified parameters and returns the response to the Verifier.

6.3 Performance tests

This section focuses on measuring the execution times of critical operations while comparing different configurations and algorithms. In particular, the point is to compare the performance of `SPHINCS-SHAKE256-SIMPLE`, `SPHINCS-SHAKE256-FAST`, `Dilithium_5` and `Mayo_5`.

1. `SPHINCS-SHAKE256-SIMPLE` and `SPHINCS-SHAKE256-FAST`: These algorithms are part of the SPHINCS+ family, and are better explained in 2.4. The SIMPLE variant emphasizes simplicity and general security, while the FAST variant focuses on faster signing times by adjusting internal parameters.
2. `Dilithium_5`: This algorithm is based on lattice-based cryptography, specifically the CRYSTALS-Dilithium scheme.
3. `Mayo_5`: Another lattice-based cryptographic algorithm that offers efficient signing and verification times. Unlike the previous ones, Mayo has not been included in the NIST standardization process.

To determine which algorithm offers better efficiency, these key metrics are analyzed:

- Keypair generation time.
- Signing Time: measures the time it takes for the Agent to sign a quote.
- Signature Verification Time: assesses the time the Verifier takes to validate the signature received from the Agent.
- Complete Attestation Cycle.

  Start: The attestation cycle begins when the Verifier sends a quote request to the Agent, generating a unique nonce for the session and a PCR mask.

  End: The cycle ends when the Verifier has successfully verified the signature and checked the IMA measurement list. In a nutshell, this parameter measures the total time required to complete an attestation interaction.
- Signature Size: the size of the signature affects the amount of data transmitted between the Agent and the Verifier.
- Public Key Size: the size of the public key impacts communication efficiency and memory consumption.
- Private Key Size.

50 measurements are taken, and the values presented represent the average time. By selecting algorithms at the same NIST security level 5, this comparison ensures that the trade-offs between performance and security remain fair and meaningful. To accurately measure execution times, timestamps are used from both the Agent and the Verifier. In particular, to capture timestamps on the Agent side, it is necessary to modify the `main.rs` file by inserting the `pretty_env_logger::init_timed()` logging mechanism, instead of the standard logger used in the `keylime_Agent`.

6.3.1 Keypair generation time

Fig. 6.1 presents the keypair generation time taken from the algorithms. Among the analyzed schemes, SPHINCS-F exhibits the fastest keypair generation time at 8 ms, significantly outperforming SPHINCS-S, which requires 48 ms, the highest value in the dataset. This difference can be attributed to the design choices and optimizations within the SPHINCS variants. For structured lattice-based algorithms, Dilithium_5 and Mayo_5 show comparable keypair generation times of 12.6 ms and 11.8 ms, respectively. These values suggest that both schemes offer efficient key generation while maintaining strong security guarantees against quantum adversaries.

6.3.2 Signing Time

Algorithm	Signing Time (ms)	NIST security level
SPHINCS-SHAKE256S-SIMPLE	481	5
SPHINCS-SHAKE256F-SIMPLE	57	5
Dilithium_5	1	5
Mayo_5	1	5

Table 6.1. Signing time comparison

The analysis reveals a significant variation in signing times across the algorithms. As we can see in 6.1, the SPHINCS+ variants focus on achieving higher security levels, but do so at the cost of longer signing times. `SPHINCS-SHAKE256S-SIMPLE` has the longest signing time, averaging 481 milliseconds, followed by `SPHINCS-SHAKE256F-SIMPLE` with a signing time of 57 milliseconds. In contrast, lattice-based algorithms `Dilithium_5` and `Mayo_5` provide near instant signing times. This level of performance makes them ideal candidates for time-critical applications where rapid signing is a priority.

6.3.3 Signature Verification Time

Fig. 6.1 shows that `SPHINCS-SHAKE256S-SIMPLE` and `SPHINCS-SHAKE256F-SIMPLE` both achieve exceptional efficiency, with an average verification time of 2 ms, demonstrating consistent and efficient performance for the verification process. `Dilithium_5` remains highly efficient, registering an average verification time of 3 milliseconds, so it still maintains a high level of performance.

`Mayo_5`, with an average time of 7 ms, is the slowest among the tested algorithms but still performs well within acceptable limits. In general, data show that verification is relatively lightweight compared to signing.

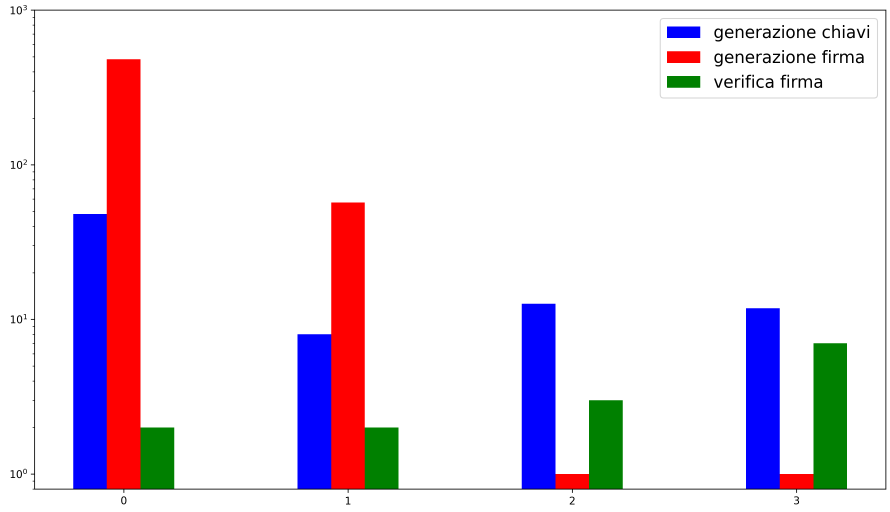

Figure 6.1. Execution Time of Key and Signature Processes

6.3.4 Complete Attestation Cycle

Table 6.2 reports the average time required to complete the attestation cycle for the evaluated post-quantum algorithms. In Keylime, a standard attestation cycle without the post-quantum

signature takes 2,5 seconds, but this value can be set to a different value through the Keylime configuration file (see file `etc/keylime.conf`). The results here demonstrate that introducing post-quantum cryptography increases the total cycle time. `SPHINCS-SHAKE256S-SIMPLE` shows the longest attestation cycle time at 2,993 ms, which is significantly impacted by its higher signing time. `SPHINCS-SHAKE256F-SIMPLE` reduces the cycle time to 2,621 ms, demonstrating the benefits of faster signing while still exceeding the baseline Keylime attestation time. Lattice-based algorithms `Dilithium_5` and `Mayo_5`, achieve the shortest attestation cycle times, measured at 2,552 ms and 2,543 ms, respectively. These values show that lattice-based approaches introduce minimal overhead while maintaining the required security guarantees.

Algorithm	Complete Attestation Cycle (ms)
SPHINCS-SHAKE256S-SIMPLE	2993
SPHINCS-SHAKE256F-SIMPLE	2621
Dilithium_5	2552
Mayo_5	2543
Without PQ algorithms	2500

Table 6.2. Complete Attestation Cycle comparison

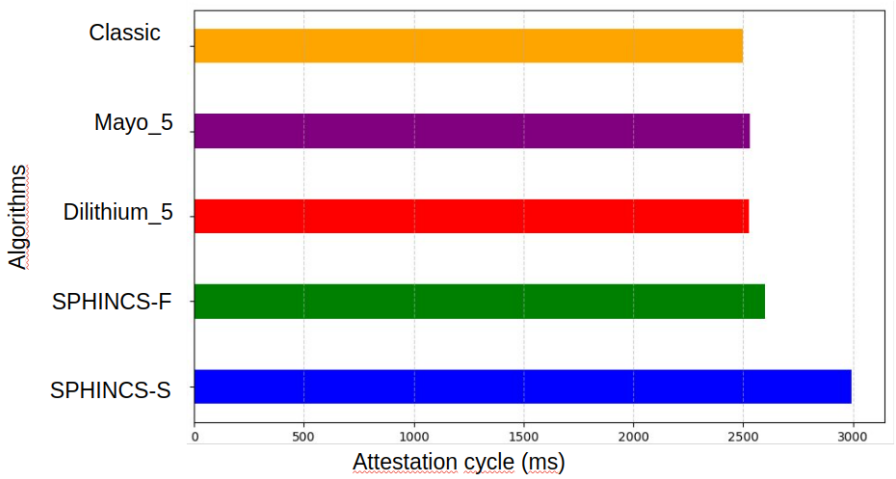

Figure 6.2. Complete Attestation Cycle comparison

6.3.5 Signature Size

Referring to fig. 6.3 SPHINCS-SHAKE256 variants produce significantly larger signatures compared to lattice-based algorithms. `SPHINCS-SHAKE256S-SIMPLE` generates a signature size of 29,792 bytes, while `SPHINCS-SHAKE256F-SIMPLE` produces signatures at 49,856 bytes. This is because the FAST variant enhances signing time at the expense of increased signature size. In contrast, the lattice-based algorithms demonstrate much smaller signature sizes. `Dilithium_5` generates signatures of 4,595 bytes, which is already an order of magnitude smaller than SPHINCS+. `Mayo_5` performs exceptionally well in this aspect, producing the smallest signature size of just 838 bytes, making it the most efficient choice in terms of minimizing communication overhead.

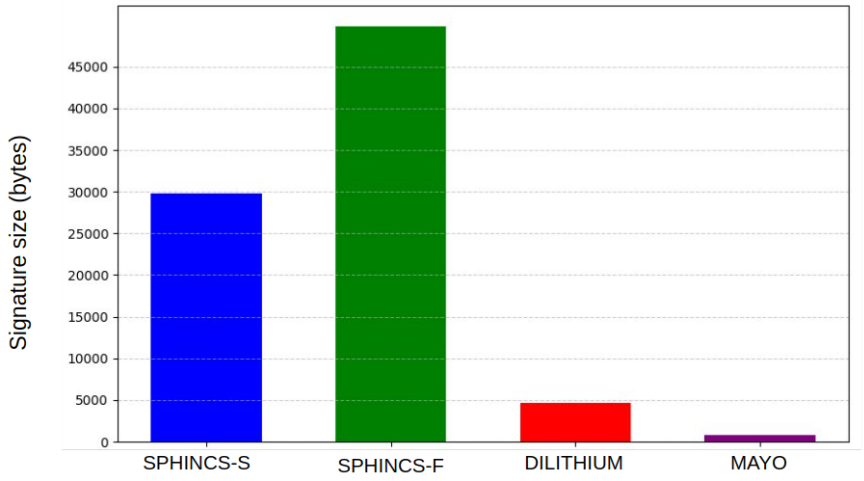

Figure 6.3. Signature size comparison

Algorithm	Public Key Size (bytes)	Private Key Size (bytes)
SPHINCS-SHAKE256S-SIMPLE	64	128
SPHINCS-SHAKE256F-SIMPLE	64	128
Dilithium_5	2592	4864
Mayo_5	5008	40

Table 6.3. Public and Private Key Sizes

6.3.6 Keypair Size

Table 6.3 presents the public key sizes for the evaluated post-quantum algorithms. Unlike signature size, public key size plays a role in how efficiently keys can be distributed and managed in constrained environments. The SPHINCS-SHAKE256 variants exhibit small public key sizes of just 64 bytes. This minimal size ensures low overhead for public key distribution. In contrast, lattice-based algorithms require significantly larger public keys:

- `Dilithium_5` has a public key size of 2,592 bytes, which is moderate compared to other lattice-based algorithms but considerably larger than SPHINCS+.
- `Mayo_5`, another lattice-based approach, requires a public key size of 5,008 bytes, making it the largest among the tested algorithms.

Taking into consideration private keys, SPHINCS-SHAKE256S-SIMPLE and SPHINCS-SHAKE256F-SIMPLE both have a relatively compact private key size of 128 bytes, benefiting from their hash-based design. On the other hand, the lattice-based Dilithium_5 exhibits the largest private key, requiring 4864 bytes, which is considerably higher than the other schemes. Mayo_5 has the smallest private key size at only 40 bytes, making it the most storage-efficient among the analyzed algorithms. The choice of algorithm will therefore depend on the specific requirements of the use case, particularly if minimizing public key size is a priority.

6.3.7 Final analysis

The integration of post-quantum digital signatures into the remote attestation process revealed notable differences among the tested algorithms. SPHINCS+ (both variants) offers strong security and small key sizes, but it comes with significant performance trade-offs. The high signature

generation time, especially in SPHINCS-SIMPLE, makes it less suitable for environments with tight real-time constraints, such as embedded systems. Although SPHINCS-FAST improves signing time, it still lags behind the lattice-based algorithms, making SPHINCS+ more appropriate for security-sensitive environments rather than time-critical applications.

In contrast, Dilithium and Mayo strike a better balance between security and efficiency. Both offer fast signing and verification times, making them well-suited for embedded systems and low-latency environments. Mayo, in particular, stands out with the smallest signature size, minimizing communication overhead, an essential factor in bandwidth-limited systems. While SPHINCS+ offers advantages in security and key size, its larger signatures and slower signing times, especially in the FAST variant, may be problematic in resource-constrained scenarios. On the other hand, Dilithium and Mayo provide smaller signature sizes and better performance. In conclusion, SPHINCS+ is suitable for applications with high-security requirements but is less optimal when performance, particularly signing time, is critical. Dilithium and Mayo are better choices for systems requiring low latency and minimal overhead. The selection of the algorithm should depend on the specific needs of the use case, balancing speed, security, and efficiency.

6.4 Resource Consumption (CPU/RAM)

This section focuses on evaluating the CPU and RAM consumption during the attestation process. These measurements are performed exclusively for the `SPHINCS-SHAKE256-SIMPLE` algorithm.

The resource usage is continuously monitored for a duration of 60 seconds while multiple attestation cycles are performed. This approach ensures that variations in CPU and RAM consumption across different stages of the attestation process are captured and represented.

The measurement is conducted using a Python script that records the resource consumption of both the Agent and the Verifier. This script runs alongside the attestation operations and uses the `psutil` library to track real-time CPU and RAM usage. The data collected includes:

- CPU Usage: This represents the percentage of processing power utilized during the attestation operations.
- RAM Usage: This measures the amount of memory consumed during the process, highlighting the memory requirements of the algorithm and the overhead introduced by the implementation.

6.4.1 RAM Consumption Analysis

Figure 6.4 represents the RAM usage during a 60-second remote attestation period. Several observations can be made:

- At the beginning of the monitoring period, the RAM consumption shows a clear increase. This corresponds to the initialization phase of the attestation process, where components such as the Verifier, Agent, or cryptographic libraries are loaded into memory.
- Then, RAM consumption stabilizes, minimally fluctuating between 36.4% and 36.8% for most of the duration. This stable state indicates that the memory requirements do not vary significantly once the attestation process starts. This behavior is expected, as the cryptographic operations and data transfers during the attestation cycle typically use a fixed memory footprint.
- Intermittent small increases in RAM consumption are observed. These could be attributed to temporary operations, such as signing or verifying data, or handling the IMA logs.

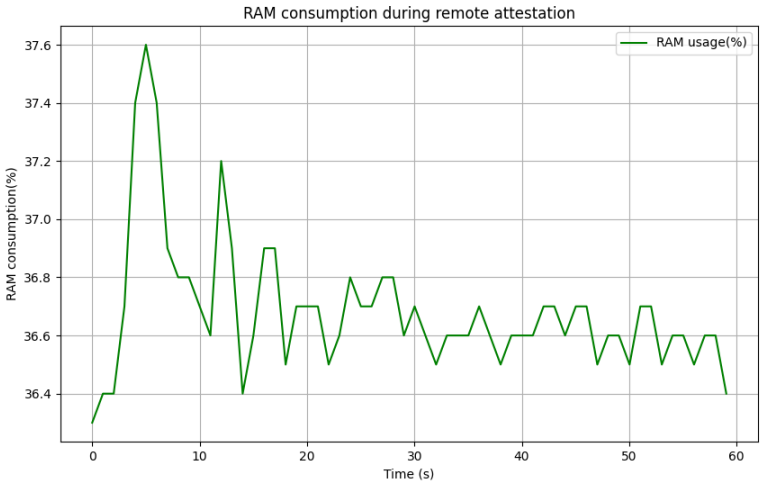

Figure 6.4. RAM usage

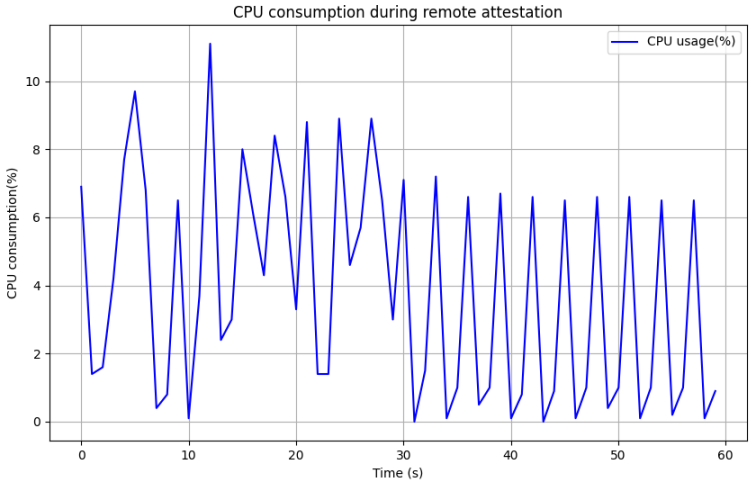

Figure 6.5. CPU usage

6.4.2 CPU Consumption Analysis

The graph 6.5 illustrates CPU usage over the same 60-second period:

- The graph shows a periodic pattern, where CPU usage increases to around 8-10% and then decreases back to near 0%. This oscillation reflects the cyclical nature of the attestation process, including phases such as quote generation, quote signing, signature verification, and IMA validation.
- The highest CPU usage values are associated with computationally intensive tasks such as generating digital signatures or verifying signatures using cryptographic algorithms.
- The low points in the graph, where CPU usage drops close to 0%, suggest that the system enters into less demanding states between attestation cycles. These intervals might represent waiting periods for responses between the Verifier and the Agent or less intensive operations like network communication.

Chapter 7

Conclusions and future work

7.1 Conclusions

In this thesis, the goal was to propose a quantum-safe solution to periodically monitor the integrity and state of nodes through remote attestation. The possibility of extending remote attestation, in particular by integrating the verification of a post-quantum digital signature, has been explored. The proposed solution enhances the security of remote attestation by incorporating quantum-resistant signature schemes, ensuring that the integrity and authenticity of the attestation process remain robust even in the face of future quantum computing advancements. The approach used involves digitally signing the quote generated by the agent within the attester. This signature is then transmitted to the verifier, which verifies both the attestation's validity and the digital signature's correctness. This mechanism ensures that any tampering attempts on the attestation data can be detected, thus strengthening the trustworthiness of the system. The implementation was carried out using Keylime, an open-source framework that enables remote attestation and supports TPM 2.0. The TPM 2.0 was leveraged as the hardware root of trust for the attestation process. To achieve post-quantum digital signing, algorithms from the liboqs library have been used, which provide foundational functions for various post-quantum cryptographic schemes. Specifically, four post-quantum signature algorithms have been evaluated: SPHINCS-FAST, SPHINCS-SIMPLE, DILITHIUM, and MAYO. These algorithms were chosen based on their distinct cryptographic properties and potential suitability. Performance testing revealed that DILITHIUM is the most efficient algorithm for the adopted solution, demonstrating superior computational performance and verification speed compared to the other evaluated algorithms. These results align with existing research indicating that structured lattice-based schemes, such as DILITHIUM, offer a good balance between security and efficiency, making them strong candidates for post-quantum authentication in real-world applications. However, this can be considered true in a software-based solution like the one adopted in this thesis. Meanwhile, in a hardware-based environment that involves modifying the TPM to support post-quantum cryptography, lattice-based algorithms are considered very difficult to manage and implement, with respect to hash-based algorithms like SPHINCS+. Lattice-based post-quantum algorithms are based on computationally intensive operations, which are not inherently optimized for execution in a TPM's constrained environment. The transition to post-quantum security is an inevitable and critical step in the evolution of cryptographic systems, particularly for security-sensitive applications such as remote attestation. The work presented in this thesis provides a foundational approach to integrating post-quantum cryptography into remote attestation and highlights key performance and implementation considerations. Future research should focus on improving efficiency, ensuring interoperability with existing TPM infrastructures, and addressing practical deployment challenges for real-world applications.

7.1.1 Future improvements and directions

One of the possible improvements in the proposed solution could be to further enhance Keylime's flexibility in supporting post-quantum cryptographic algorithms. A useful extension would be to

provide users with the ability to select from a range of post-quantum algorithms and customize parameters such as public and private keys, enabling greater adaptability to specific security and performance requirements. The more ambitious long-term goal for achieving quantum-safe remote attestation would be the integration of post-quantum cryptographic capabilities directly into the TPM. This would allow the TPM itself to support post-quantum algorithms natively, generate the key pair required for digital signing and verification at manufacturing time, and directly sign the quote using a post-quantum private key. Such an enhancement would improve security by minimizing external dependencies and ensuring that key management remains within a trusted hardware boundary. However, achieving this goal requires significant advancements in TPM hardware and firmware, as the TPM is a constrained environment with limited computational power, memory, and specific hardware design constraints that make the adoption of certain cryptographic primitives more challenging. The work contributes to the development of hybrid cryptographic approaches that combine classical and post-quantum signatures to ensure a smooth and secure transition to quantum-resistant attestation mechanisms. This approach could mitigate potential vulnerabilities that may arise during the initial adoption phase of post-quantum cryptographic schemes.

Bibliography

[1] K.Townsend, "Solving the Quantum Decryption 'Harvest Now, Decrypt Later' Problem", SecurityWeek, vol. 28, February 2022, p. 5. https://www.securityweek.com/solving-quantum-decryption-harvest-now-decrypt-later-problem

[2] M.Kumar, "Post-quantum cryptography Algorithm's standardization and performance analysis", Array, August 2022, p. 27, DOI 10.1016/j.array.2022.100242

[3] R.Feynman, "Simulating physics with computers", International Journal of Theoretical Physics, vol. 21, June 1982, p. 21, DOI 10.1007/BF02650179

[4] V.Mavroiedis, K.Vishi, M.Zych, and A.Josang, "The Impact of Quantum Computing on Present Cryptography", International Journal of Advanced Computer Science and Applications, vol. 9, no. 3, 2018, p. 10, DOI 10.14569/IJACSA.2018.090354

[5] S.Yasmineh, "Foundations of Quantum Mechanics", Encyclopedia, vol. 2, May 2022, pp. 1082–1090, DOI 10.1109/MS.2010.160

[6] E.Chae, J.Choi, and J.Kim, "An elementary review on basic principles and developments of qubits for quantum computing", Nano Convergence, vol. 11, March 2024, p. 13, DOI 10.1186/s40580-024-00418-5

[7] Y.Yu, S.Zhu, G.Sun, X.Wen, N.Dong, J.Chen, P.Wu, and S.Han, "Quantum Jumps between Macroscopic Quantum States of a Superconducting Qubit Coupled to a Microscopic Two-Level System", Physical Review Letters, vol. 101, October 2008, p. 4, DOI 10.1103/physrevlett.101.157001

[8] P.Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", SIAM Journal on Computing, vol. 26, October 1997, pp. 1484–1509, DOI 10.1137/s0097539795293172

[9] L.Grover, "A fast quantum mechanical algorithm for database search", 28th Annual ACM Symposium on Theory of Computing (STOC), Philadelphia (Pennsylvania), May 29-30, 1996, pp. 212–219, DOI 10.48550/arXiv.quant-ph/9605043

[10] R. J. Anderson, "Security engineering", Wiley, 2008

[11] X.Liu, D.Xiao, and C.Liu, "Double Quantum Image Encryption Based on Arnold Transform and Qubit Random Rotation", Entropy, vol. 20, November 2018, p. 16, DOI 10.3390/e20110867

[12] Z.Liao, Q.Huang, and X.Chen, "A fully dynamic forward-secure group signature from lattice", Cybersecurity, vol. 5, October 2022, p. 14, DOI https://doi.org/10.1186/s42400-022-00122-z

[13] B.Biswas and N.Sendrier, "Mceliece cryptosystem implementation: Theory and practice", Post-Quantum Cryptography, Berlin (Germany), 2008, pp. 47–62, DOI 10.1007/978-3-540-88403-3_4

[14] A.Valentijn, "Goppa Codes and Their Use in the McEliece Cryptosystems", Honors Capstone Project, vol. 4, January 2015, p. 41. https://surface.syr.edu/honors_capstone/845

[15] M.Mosca, "Cybersecurity in an Era with Quantum Computers: Will We Be Ready?", IEEE Security & Privacy, vol. 16, September 2018, p. 14, DOI 10.1109/MSP.2018.3761723

[16] D.Joseph1, R.Misoczki, M.Manzano1, J.Tricot, F.Dominguez, S. O.Lacombe, J.Hidary, P.Venables, and R.Hansen, "Transitioning organizations to post-quantum cryptography", Nature, vol. 605, May 2022, p. 5, DOI 10.1038/s41586-022-04623-2

[17] D.Mayers, "Unconditional security in Quantum Cryptography", Journal of Applied and Computational Mechanics, vol. 48, May 2001, pp. 351–406, DOI 10.48550/arXiv.quant-ph/9802025

[18] C.Bennett and G.Brassard, "Quantum cryptography: Public key distribution and coin tossing", Theoretical Computer Science, vol. 560, December 2014, pp. 7–11, DOI 10.1016/j.tcs.2014.05.025

[19] W.Buchanan and A.Woodward, "Will quantum computers be the end of public key encryption?", Journal of Cyber Security Technology, vol. 1, September 2016, pp. 1–22, DOI 10.1080/23742917.2016.1226650

[20] R.Merkle, "A certified digital signature", Advances in Cryptology — CRYPTO' 89 Proceedings (G.Brassard, ed.), pp. 218–238, Springer, 1990, DOI 10.1007/0-387-34805-0_21

[21] M.Campagna, L.Chen, O.Dagdelen, J.Ding, J.Fernick, N.Gisin, D.Hayford, T.Jennewein, N.Lutkenhaus, M.Mosca, B.Neill, M.Pecen, R.Perlner, G.Ribordy, J.Schank, D.Stebila, N.Walenta, W.Whyte, and Z.Zhang, "Quantum Safe Cryptography and Security", ETSI, vol. 15, Jube 2015, p. 64. https://www.etsi.org/images/files/etsiwhitepapers/quantumsafewhitepaper.pdf

[22] T.Moriya, K.Takashima, and T.Takagi, "Group Key Exchange from CSIDH and Its Application to Trusted Setup in Supersingular Isogeny Cryptosystems", Information Security and Cryptology, March 2020, pp. 86–98, DOI 10.1007/978-3-030-42921-8_5

[23] A.Huelsing, D.Butin, S.Gazdag, J.Rijneveld, and A.Mohaisen, "XMSS: eXtended Merkle Signature Scheme." RFC-8391, May 2018, DOI 10.17487/RFC8391

[24] D.McGrew, M.Curcio, and S.Fluhrer, "Leighton-Micali Hash-Based Signatures." RFC-8554, April 2019, DOI 10.17487/RFC8554

[25] D.Dam, T.Tran, V.Hoang, C.Pham, and T. Hoang, "A Survey of Post-Quantum Cryptography: Start of a New Race", Cryptography, vol. 7, July-August 2023, p. 18, DOI 10.3390/cryptography7030040

[26] W.Beullens, "Breaking rainbow takes a weekend on a laptop", 'Advances in Cryptology - CRYPTO 2022: 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part II (T. Y.Dodis, ed.), pp. 464–479, Springer, 2022, DOI 10.1007/978-3-031-15979-4_16

[27] Keyfactor, https://www.keyfactor.com/blog/certificate-chain-of-trust/

[28] K.Ezirim, W.Khoo, G.Koumantaris, and R.Law, "Trusted Platform Module - A Survey", ResearchGate, November 2012, p. 14. https://www.researchgate.net/publication/287984174_Trusted_Platform_Module_-_A_Survey

[29] Trusted Computing Group, https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary/

[30] A.Tomlinson, "Smart cards, tokens, security and applications", Springer, 2008

[31] G.Coker, J.Guttman, P.Loscocco, A.Herzog, J.Millen, B.O'Hanlon, J.Ramsdell, A.Segall, J.Sheehy, and B.Sniffen, "Principles of remote attestation", International Journal of Information Security, vol. 10, April 2011, pp. 63–81, DOI 10.1007/s10207-011-0124-7

[32] T. Dierks and E. Rescorla, "H.Birkholz, D.Thaler, M.Richardson, N.Smith, W.Pan." RFC-9334, January 2023, DOI 10.17487/RFC9334

[33] N. Schear, P. T. Cable, T. M. Moyer, B. Richard, and R. Rudd, "Bootstrapping and maintaining trust in the cloud", Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 65–77, Association for Computing Machinery, 2016, DOI 10.1145/2991079.2991104

Appendix A

User's manual

A.1 Linux Installation

The first step is to install the Ubuntu Linux operating system by following the step-by-step instructions provided below. This requires a USB drive (at least 8 GB) and a program to create a bootable drive, such as Rufus.

A.1.1 Download the Ubuntu ISO image

Visit Ubuntu's official website https://www.ubuntu-it.org/download and download the ISO for Ubuntu 24.04.1 LTS. Insert a USB drive into your machine and launch Rufus, available from the official website https://rufus.ie/it/, then:

- Select the downloaded ISO image.
- Choose the USB drive as the destination.
- Click Start to create the bootable USB.

A.1.2 Configure the BIOS/UEFI and begin Installation

Access the BIOS/UEFI settings by restarting the PC and pressing the appropriate key during startup (e.g., F2, F12, DEL, or ESC). Set the USB drive as the boot device by adjusting the boot order in the BIOS/UEFI settings to prioritize it. Insert the bootable USB drive into the computer and restart it. The system will load the Ubuntu installer. Set "Try or install Ubuntu". Choose the preferred language (e.g., English) and click "install Ubuntu". When prompted to select the installation type, you can choose:

- Install Ubuntu alongside Windows Boot Manager.
- Something else for manual partitioning. This is the one to select if you want to configure the PC in dual-boot.

A.1.3 Configure the Disk

This section applies only to cases where the system is configured for dual-boot, requiring a partitioning method selection. If you are not setting up a dual-boot system, you can skip this part.

If you selected "Something else", the system's actual partitions will be displayed. Select "Free Space", and then you can add the first partition, which will be dedicated to the new operating

system. Select the necessary size for your partition, specifically choosing the *total_available* − *total_of_swap_partition*. in this case, the swap partition will be 4 GB. Then, you have to create another partition dedicated to the swap, with a size of 4 GB.

Root Partition:

- Type: Ext4.
- Size: At least 25 GB.

Swap Partition:

- Type: Swap area.
- Size: 4 GB.

Confirm and write changes to the disk, allowing the machine to install the Bootloader. During installation, ensure the bootloader (GRUB) is installed on the correct drive (e.g., `/dev/sda` for the main disk). Then press "Restart now" to reboot the system. Upon reboot, `GRUB` will appear, allowing you to choose between Windows and Ubuntu. Boot into each operating system to confirm everything works correctly.

A.2 Keylime Installation

First of all, install the following dependencies:

```
$ sudo apt install libssl-dev swig python3-pip
```

The Keylime framework needs a version of libtss2 >= 2.4.0, and since Ubuntu 20.04 has by default the version 2.3.2, you need first to uninstall it:

```
$ sudo apt remove libtss2-esys0

$ sudo apt autoclean \&\& sudo apt autoremove
```

Now you can manually build and install libtss2 version >= 2.4.0. However, first, install the following dependencies:

```
$ sudo apt install autoconf autoconf-archive libglib2.0-dev libtool
pkg-config libjson-c-dev libcurl4-gnutls-dev
```

Now you can install all the tools required to manage the TPM 2.0 chip.

- Installation of `libtss2`

```
$ git clone https://github.com/tpm2-software/tpm2-tss.git
$ cd tpm2-tss
$ ./bootstrap
$ ./configure --prefix=/usr
$ make
$ sudo make install
```

- Installation of `tpm2-tools`

```
$ git clone https://github.com/tpm2-software/tpm2-tools.git
$ cd tpm2-tools
$ ./bootstrap
$ ./configure --prefix=/usr/local
$ make
$ sudo make install
```

- Installation of `tpm2-abrmd`

```
$ git clone https://github.com/tpm2-software/tpm2-abrmd.git
$ cd tpm2-abrmd
$ ./bootstrap
$ ./configure --with-dbuspolicydir=/etc/dbus-1/system.d \
    --with-systemdsystemunitdir=/lib/systemd/system \
    --with-systemdpresetdir=/lib/systemd/system-preset \
    --datarootdir=/usr/share
$ make
$ sudo make install
$ sudo ldconfig
$ sudo pkill -HUP dbus-daemon
$ sudo systemctl daemon-reload
```

Configure TPM Command Transmission Interface (TCTI):

```
$ export TPM2TOOLS_TCTI="tabrmd:bus_name=com.intel.tss2.Tabrmd"
```

Start the Access Broker Resource Manager service:

```
$ sudo service tpm2-abrmd start
```

And check if it is working:

```
systemctl status tpm2-abrmd.service
```

You should read "active (running)". To check if the tpm2 tools are working properly, you can run the command:

```
$ tpm2_pcrread
```

This will display the content of the PCR banks. Now, you can proceed by installing the framework. First, clone the Keylime repository, move into the "keylime" directory, install the script, and copy the configuration file:

```
$ git clone https://github.com/keylime/keylime.git
$ cd keylime
$ sudo pip3 install . -r requirements.txt
$ sudo cp keylime.conf /etc/
```

A.2.1 Keylime Agent Configuration

This section describes how to configure the Keylime Agent on the attester machine. It is important to underline that the Python Agent present in Keylime will be deprecated. However, the workflow for setting it up and using it properly is still provided here. The configuration of the Rust-Agent, which is the one used in this work, is presented in A.3.

To configure the Python Agent on the attester machine, open the configuration file:

```
sudo nano /etc/keylime.conf
```

The file is divided into sections, each containing several parameters. For the Agent configuration, you need to modify only the sections `[general]` and the `[cloud agent]`. In the `[general]` section, find the receive revocation ip parameter and put the IP address of the attester machine, for example:

```
receive_revocation_ip = 192.168.0.100
```

In the `[cloud agent]` section set the cloud agent ip which is the IP address of the attester machine, the registrar ip which is the IP address of the registrar, and the `agent uuid`, for example:

```
cloud_agent_ip = 192.168.0.100
registrar_ip = 192.168.0.114
agent_uuid = d432fbb3-d2f1-4a97-9ef7-75bd81c00000
```

Note that the `agent uuid` shown in the example is the default one. Then set the hash algorithm:

```
tpm_hash_alg = sha256
```

A.2.2 Keylime Verifier and Registrar configuration

In this section, we will configure the Keylime Verifier, Keylime Registrar, and Keylime Tenant components. The tests are performed by running these three components on the same machine, although they may be installed and run on different machines. The machine has to be equipped with a TPM 2.0 because these components use the tpm2-tools for performing some operations; differently from the attester machine, which needs a hardware TPM, the TPM 2.0 installed on this machine may also be an emulator as we only need the functionalities provided by the tpm2-tools. If you have an attester machine containing the Rust Agent and a Verifier machine that includes the Verifier, Registrar, and Tenant, install Keylime by following the instructions in section A.2, then customize the Keylime configuration file as described in the sections below. Otherwise, if your components are all on the same machine (as in this work) and you have already installed Keylime, you can customize the configuration files directly.

A.2.3 Registrar

Open the configuration file:

```
$ sudo nano /etc/keylime.conf
```

Find the `[registrar]` tag. Then customize the registrar ip by putting the IP address of the verifier machine, for example:

The tests are performed, leaving the default values for the other parameters.

```
registrar_ip = 192.168.0.114
```

A.2.4 Verifier

Open the configuration file:

```
$ sudo nano /etc/keylime.conf
```

and find the `[cloud verifier]` tag. Then customize the cloudverifier ip, the registrar ip, and the revocation notifier ip, by putting the IP address of the verifier machine, for example:

```
cloudverifier_ip = 192.168.0.114
registrar_ip = 192.168.0.114
revocation_notifier_ip = 192.168.0.114
```

The tests are performed leaving the default values for the other parameters.

A.2.5 Tenant

Open the configuration file:

```
$ sudo nano /etc/keylime.conf
```

and find the `[tenant]` tag. Then customize the cloudverifier ip and the registrar IP by putting the IP address of the verifier machine, for example:

```
cloudverifier_ip = 192.168.0.114
registrar_ip = 192.168.0.114
```

Then, from the attester machine retrieve the PCRs values with indexes 0-9 of the SHA256 bank, through the command:

```
$ sudo tpm2_pcrread
```

and copy them in the tpm policy parameter in one line, as a JSON object, like:

```
tpm_policy = {"0": ["47D..."], "1":["25C..."], ..., "9":["4C3..."]}
```

Finally, go into the Keylime directory and copy the content of the `tpm_cert_store` directory in `/var/lib/keylime/tpm_cert_store`:

```
$ sudo mkdir /var/lib/keylime/tpm_cert_store
$ sudo cp -r ./tpm_cert_store /var/lib/keylime/tpm_cert_store
```

A.3 Rust implementation of Keylime Agent: installation

This section provides a comprehensive guide to installing the Rust implementation of the Keylime agent. The rust-keylime agent is the official agent (starting with version 0.1.0) and replaces the Python implementation.

A.3.1 Prerequisites

Before proceeding with the installation, ensure that your system meets the necessary prerequisites. This includes installing required packages and setting up the Rust programming environment. The required packages vary based on your operating system. Below are the instructions for Fedora, Debian, and Ubuntu systems.

Fedora

For Fedora systems, the following packages are essential for building the Rust-Keylime agent:

```
$ sudo dnf install clang openssl-devel tpm2-tss-devel zeromq-devel
```

For runtime requirements, install the following packages:

```
$ sudo dnf install openssl tpm2-tss systemd util-linux-core zeromq
```

Debian and Ubuntu

For Debian-based systems, install the necessary development packages using:

```
$ sudo apt-get install libclang-dev libssl-dev libtss2-dev libzmq3-dev
    pkg-config
```

For runtime dependencies, execute:

```
$ sudo apt-get install coreutils libssl1.1 libtss2-esys-0 systemd libzmq3
```

A.3.2 Installing Rust

Ensure that Rust is installed on your system before proceeding. You can install Rust by following the instructions in the official Rust website. To verify the installation, run:

```
$ rustc --version
```

A.3.3 Cloning the Rust-Keylime Repository

Begin by cloning the official Rust-Keylime repository from GitHub:

```
$ git clone https://github.com/keylime/rust-keylime.git
$ cd rust-keylime
```

With all prerequisites in place, you can now build the Rust-Keylime agent.

```
$ cargo build --release
```

A.3.4 Configuring Logging

To enable detailed logging-in for troubleshooting and monitoring, set the `RUST\_LOG` environment variable before running the agent. For example, to activate trace-level logging:

```
$ export RUST_LOG=keylime_agent=trace
$ cargo run --bin keylime_agent
```

To ensure that the installation is successful and the agent works as expected, run the unit tests provided in the repository:

```
$ cargo test
```

A.3.5 Deploying the Agent as a systemd Service

For seamless deployment and management, the Rust-Keylime agent can be run as a systemd-managed service. Follow these steps to setup it:

```
$ make
$ sudo make install
$ sudo systemctl start keylime_agent
$ sudo systemctl enable keylime_agent
```

A.3.6 Building a Debian Package with cargo-deb

For Debian-based systems, you can create a Debian package using `cargo-deb`. This simplifies the installation and distribution process. First, ensure that Rust is updated, then install `cargo-deb`:

```
$ rustup update
$ cargo install cargo-deb
```

Navigate to the Rust-Keylime project directory and execute:

```
$ cargo deb -p keylime_agent
```

This command generates a .deb package for the Keylime agent, which can be installed using `dpkg` or `apt`.

After completing the installation and configuration, verify that the Rust-Keylime agent is running correctly by checking its status:

```
$ sudo systemctl status keylime_agent
```

A.4 How to use Keylime

A.4.1 Basic commands

After the setup, some commands become available for use from the command line:

- keylime_verifier: start the verifier service;
- keylime_tenant: start the tenant service;
- keylime_userdata_encrypt: encryption of a given file;
- keylime_registrar: start the registrar service;
- keylime_ca: handle the certification authority;
- keylime_attest: verification of the state of all the agents registered in the persistence storage;
- keylime_convert_runtime_policy: for the runtime policy conversion;
- keylime_sign_runtime_policy: signs Keylime runtime policies using DSSE;
- keylime_upgrade_config: parses the content of a configuration file and uses the data to replace the values in templates to generate new configuration files;
- keylime_create_policy: create a JSON allowlist/policy from given files as input;
- keylime_agent: start the python agent service (deprecated).

A.4.2 Keylime runtime policies

A runtime policy is a collection of "golden" cryptographic hashes of files' untampered state or of keys that may be loaded onto keyrings for IMA verification. Keylime will load the runtime policy into the Keylime Verifier. Keylime will then poll the tpm quotes with PCR 10 on the agents' TPM and validate the state of the agents' file(s) against the policy. If the object has been tampered with or an unexpected key was loaded onto a keyring, the hashes will not match and Keylime will place the agent into a failed state. Likewise, if any files invoke the actions stated in ima-policy that are not matched in the allowlist, keylime will set the agent into a failed state.

Generate a Runtime Policy

Runtime policies depend on the IMA configuration and used files by the operating system. Keylime provides two helper scripts for getting started. The first script generates a runtime policy from the initramfs, IMA log (just for the boot aggregate), and files located on the root filesystem of a running system. The `create_runtime_policy.sh` script is available here.

Run the script as follows:

```
# create_runtime_policy.sh -o runtime_policy_keylime.json
```

For more options, see the help page `create\_runtime\_policy.sh`:

```
Usage: $0 -o/--output_file FILENAME [-a/--algo ALGO]
    [-x/--ramdisk-location PATH] [-y/--boot_aggregate-location PATH]
    [-z/--rootfs-location PATH] [-e/--exclude_list FILENAME]
    [-s/--skip-path PATH]"

optional arguments:
-a/--algo (checksum algorithmi to be used, default: sha1sum)
-x/--ramdisk-location (path to initramdisk, default: /boot/, set to
    "none" to skip)
```

```
-y/--boot_aggregate-location (path for IMA log, used for boot aggregate
    extraction, default:
    /sys/kernel/security/ima/ascii_runtime_measurements, set to "none" to
    skip)
-z/--rootfs-location (path to root filesystem, default: /, cannot be
    skipped)
-e/--exclude_list (filename containing a list of paths to be excluded
    (i.e., verifier will not try to match checksums), default: none)
-s/--skip-path (comma-separated path list, files found there will not
    have checksums calculated, default: none)
-h/--help show this message and exit
```

The resulting `runtime_policy_keylime.json` file can be directly used by the `keylime_tenant` (option –runtime-policy)

Creating more Complex Policies

The second script allows users to build more complex policies by providing options to include keyring verification, IMA verification keys, generate an allowlist from the IMA measurement log, and extending existing policies. A basic policy can be easily created by using the IMA measurement log from the system:

```
$ keylime_create_policy -m /path/to/ascii_runtime_measurements -o
    runtime_policy.json
```

For more options, see the help page `keylime_create_policy` -h:

```
usage: keylime_create_policy [-h] [-B BASE_POLICY] [-k] [-b] [-a
    ALLOWLIST] [-m IMA_MEASUREMENT_LIST] [-i IGNORED_KEYRINGS] [-o
    OUTPUT] [--no-hashes] [-A IMA_SIGNATURE_KEYS]

options:
  -h, --help show this help message and exit
  -B BASE_POLICY, --base-policy BASE_POLICY
                        Merge new data into the given JSON runtime policy
  -k, --keyrings Create keyrings policy entries
  -b, --ima-buf Process ima-buf entries other than those related to
      keyrings
  -a ALLOWLIST, --allowlist ALLOWLIST
                        Use given plain-text allowlist
  -m IMA_MEASUREMENT_LIST, --ima-measurement-list IMA_MEASUREMENT_LIST
                        Use given IMA measurement list for keyrings and
                            critical data extraction rather than
                            /sys/kernel/security/ima/ascii_runtime_measurements
  -i IGNORED_KEYRINGS, --ignored-keyrings IGNORED_KEYRINGS
                        Ignored the given keyring; this option may be
                            passed multiple times
  -o OUTPUT, --output OUTPUT
                        File to write JSON policy into; default is to print
                            to stdout
  --no-hashes Do not add any hashes to the policy
  -A IMA_SIGNATURE_KEYS, --add-ima-signature-verification-key
      IMA_SIGNATURE_KEYS
                        Add the given IMA signature verification key to the
                            Keylime-internal 'tenant_keyring'; the key
                            should be an x509 certificate in DER or PEM
                            format but may also be a public or private key
                        file; this option may be passed multiple times
```

Remotely Provision Agents

Now that our runtime policy is available, we can send it to the verifier. Using the `keylime_tenant` we can send the runtime policy as follows:

```
$ keylime_tenant -c add --uuid <agent-uuid> --runtime-policy
    /path/to/policy.json
```

You can use "-c update" if your agent is already registered.

To test this, create a script that does anything (for example echo "hello world") that is not present in your runtime policy. Run the script as root on the agent. You will then see the following output on the verifier showing the agent status change to failed:

```
keylime.tpm - INFO - Checking IMA measurement list...
keylime.ima - WARNING - File not found in allowlist: /root/evil_script.sh
keylime.ima - ERROR - IMA ERRORS: template-hash 0 fnf 1 hash 0 good 781
keylime.cloudverifier - WARNING - agent
    D432FBB3-D2F1-4A97-9EF7-75BD81C00000 failed, stopping polling
```

A.4.3 Keylime CLI

In this section are the main commands offered by the Keylime Command-Line Interface to interact with the framework. Each command respects the format:

```
$ keylime_tenant -c [command]
```

where the -c option can take one of the following keywords.

Add

The `Add` command is used to register an agent with the Verifier. It enables periodic remote attestation and can take the following parameters:

- -u [UUID]

Where [UUID] is the agent UUID to be added. If not specified, the default UUID is used, which corresponds to $d432fbb3 - d2f1 - 4a97 - 9ef7 - 75bd81c00000$.

- -v [Verifier IP]

Where [Verifier IP] is the IP address of the Verifier where the agent has to be registered. If not specified, is used the IP address inserted into:

`/etc/keylime.conf.`

- -t [Agent IP]

Where [Agent IP] is the IP address of the agent to be added.

- -f [payload]

Where [payload] is a file to be encrypted with the bootstrap key.

- -exclude [exclude list]

Where [exlude list] is the file containing a regular expression with the files to be excluded in the validation process of the Measurement List.

- -allowlist [whitelist]

Where [whitelist] is the file containing the golden values used to validate ML entries.

Update

The `Update` command updates an already registered agent. The parameters are the same of the `Add` command. An example of this command is:

```
$ sudo keylime_tenant -c update -u d432fbb3-d2f1-4a97-9ef7-75bd81c00000
    -t 192.168.0.100 -f payload --exclude exclude_host --allowlist
    whitelist --pods_list pods_list
```

Delete

The `Delete` command removes a registered Agent from the Verifier. It can take parameters:

- -u [UUID]

Where [UUID] is the agent UUID to be removed. If not specified, the default UUID is used, which corresponds to $d432fbb3 - d2f1 - 4a97 - 9ef7 - 75bd81c00000$.

- -v [Verifier IP]

Where [Verifier IP] is the IP address of the Verifier machine where the agent has to be deleted. If not specified, is used the IP address inserted into:

`/etc/keylime.conf.`

An example of this command is:

```
$ sudo keylime_tenant -c delete -u d432fbb3-d2f1-4a97-9ef7-75bd81c00000
```

Status

The `Status` command allows the user to retrieve the status of a registered agent. An example of this command is:

```
$ sudo keylime_tenant -c status -u [UUID]
```

where the placeholder [UUID] represents the agent UUID. If not specified, the default UUID is used, which corresponds to $d432fbb3 - d2f1 - 4a97 - 9ef7 - 75bd81c00000$. Optionally, if more than one Verifier is used, the specific verifier IP address can be specified through the option:

```
$ sudo keylime_tenant -c status -u [UUID] -v [Verifier_IP]
```

A.5 Installing Liboqs

This section explains how to install and configure liboqs, a C library for post-quantum cryptography. Liboqs is developed by the Open Quantum Safe (OQS) project to provide a collection of post-quantum cryptographic algorithms.

A.5.1 Prerequisits

Before starting, ensure your system meets the following prerequisites:

- a C compiler (e.g., GCC, Clang, or MSVC).
- Git (for cloning the repository).
- CMake (version 3.14 or later).

A.5.2 Install required dependencies

For Ubuntu/Debian:

```
$ sudo apt-get update
$ sudo apt-get install cmake gcc libssl-dev
```

For Fedora:

```
$ sudo dnf install cmake gcc openssl-devel
```

For macOS: Install the required tools using Homebrew:

```
$ brew install cmake openssl
```

For Windows:

- Install CMake and Visual Studio (Community Edition is sufficient).
- Ensure openssl is installed or build it from source if needed.

A.5.3 Clone the liboqs Repository

Download the latest version of the liboqs source code from the official GitHub repository:

```
$ git clone --recursive https://github.com/open-quantum-safe/liboqs.git
```

The `--recursive` option ensures that submodules are initialized and cloned along with the main repository.

A.5.4 Build the library

Navigate to the liboqs directory, create a build directory, and navigate into it. Then, run `CMake` to configure the build, and finally build and install the library:

```
$ cd liboqs
$ mkdir build && cd build
$ cmake -DCMAKE_INSTALL_PREFIX=<install_path> ..
$ make -j$(nproc)
$ sudo make install
```

A.5.5 Verify the Installation

To ensure the library is correctly installed, check if the shared library is present in the specified install directory (e.g., /usr/local/lib). Test the library by running the example programs included in the repository:

```
$ ./tests/test_kem
$ ./tests/test_sig
```

Appendix B

Developer's manual

This section details the modifications made to Keylime to add post-quantum cryptographic support. Specifically, it explains the integration of post-quantum signature verification into the remote attestation process.

B.1 Configuration of the IMA policy

Keylime's runtime integrity monitoring requires the setup of Linux IMA. Create a file `/etc/ima/ima-policy` with the following content:

```
# PROC_SUPER_MAGIC
dont_measure fsmagic=0x9fa0
# SYSFS_MAGIC
dont_measure fsmagic=0x62656572
# DEBUGFS_MAGIC
dont_measure fsmagic=0x64626720
# TMPFS_MAGIC
dont_measure fsmagic=0x01021994
# RAMFS_MAGIC
dont_measure fsmagic=0x858458f6
# SECURITYFS_MAGIC
dont_measure fsmagic=0x73636673
# SELINUX_MAGIC
dont_measure fsmagic=0xf97cff8c
# CGROUP_SUPER_MAGIC
dont_measure fsmagic=0x27e0eb

measure func=BPRM_CHECK mask=MAY_EXEC
measure func=FILE_MMAP mask=MAY_EXEC
```

B.1.1 Configuring IMA Appraisal for File Integrity Verification

Enable IMA Appraisal in Linux. To do it, go to `/etc/default/grub` and append the following parameters to the `GRUB\_CMDLINE\_LINUX\_DEFAULT`:

```
quiet splash ima_policy=tcb
```

Then, exit, update changes, and reboot the system.

```
sudo update-grub
reboot
```

Now, the correct policy can be created with the command:

```
keylime_create_policy -m /path/to/ascii_runtime_measurements -o
    runtime_policy.json
```

B.2 Keylime Agent modifications

B.2.1 Main.rs

In file `main.rs` add the declaration of the function in charge of generating the post-quantum keys:

```
use libc::size_t;

#[link(name = "gen_keypair")]
extern "C" {
    fn generate_PQ_keypair(stampa: i32) -> KeypairResult;
}
```

And the prototype of the function's return value:

```
#[repr(C)]
struct KeypairResult {
    public_key: *const u8,
    public_key_len: size_t,
    private_key: *const u8,
    private_key_len: size_t,
}
```

Modify the struct `QuoteData` by adding the highlighted fields related to the created keys:

```
pub struct QuoteData {
    tpmcontext: Mutex<tpm::Context>,
    priv_key: PKey<Private>,
    pub_key: PKey<Public>,
    ak_handle: KeyHandle,
    payload_tx: mpsc::Sender<payloads::PayloadMessage>,
    revocation_tx: mpsc::Sender<revocation::RevocationMessage>,
    keys_tx: mpsc::Sender<(
        keys_handler::KeyMessage,
        Option<oneshot::Sender<keys_handler::SymmKeyMessage>>,
    )>,
    hash_alg: keylime::algorithms::HashAlgorithm,
    enc_alg: keylime::algorithms::EncryptionAlgorithm,
    sign_alg: keylime::algorithms::SignAlgorithm,
    agent_uuid: String,
    allow_payload_revocation_actions: bool,
    secure_size: String,
    work_dir: PathBuf,
    ima_ml_file: Option<Mutex<fs::File>>,
    measuredboot_ml_file: Option<Mutex<fs::File>>,
    ima_ml: Mutex<MeasurementList>,
    secure_mount: PathBuf,
    pq_pub_key: Vec<u8>,
    pq_pub_key_len: usize,
    pq_priv_key: Vec<u8>,
    pq_priv_key_len: usize,
}
```

In function `main.rs`, add the code needed for calling the function of keypair generation and for the conversion of key types:

```
let mut pq_key_base64 = String::new();

    let pq_result = unsafe {
        generate_PQ_keypair(1)
    };

    let public_key_vec = unsafe {
        if !pq_result.public_key.is_null() {
            Vec::from(std::slice::from_raw_parts(
                pq_result.public_key,
                pq_result.public_key_len,
            ))
        } else {
            Vec::new() // Restituisce un vettore vuoto se il puntatore Ã¨
                nullo
        }
    };

    let private_key_vec = unsafe {
        if !pq_result.private_key.is_null() {
            Vec::from(std::slice::from_raw_parts(
                pq_result.private_key,
                pq_result.private_key_len,
            ))
        } else {
            Vec::new()
        }
    };
    let pq_key_base64: String =
        general_purpose::STANDARD.encode(&public_key_vec);

    unsafe {
        libc::free(pq_result.public_key as *mut _);
    }
```

creation of the struct `Quotedata`:

```
 let quotedata = web::Data::new(QuoteData {
   tpmcontext: Mutex::new(ctx),
   priv_key: nk_priv,
   pub_key: nk_pub,
   ...
   ...
   pq_pub_key: public_key_vec,
   pq_pub_key_len: pq_result.public_key_len,
   pq_priv_key: private_key_vec,
   pq_priv_key_len: pq_result.private_key_len,
});
```

and insert the code to perform the signature of the response sent in the registration phase:

```
// Conversione del campo auth_tag in CString
   let auth_tag_ptr: *const u8 = auth_tag.as_ptr() as *const u8;

   //conversione chiave privata in puntatore
   let pq_priv_key_cstring: *const u8 = private_key_vec.as_ptr();
```

```
        // Chiamata alla funzione C
        let signed_response = unsafe {
        sign_with_PQ(auth_tag_ptr, auth_tag.len(),pq_priv_key_cstring,
            pq_result.private_key_len)
        };

        let signature_slice = unsafe {
            std::slice::from_raw_parts(signed_response.signature,
            signed_response.signature_len as usize) };
        let signature_vec = signature_slice.to_vec();
```

To add the verification of the post-quantum public key, extend the struct `Activate` in file `registrar_agent.rs`:

```
    struct Activate<'a> {
    auth_tag: &'a str,
    auth_tag_len: usize,
    sign_pq: Vec<u8>,
    sign_pq_len: c_ulong,
    pq_key: Vec<u8>,
    pq_key_len: usize,
}
```

B.2.2 Function for post-quantum keypair generation

Content of the file `gen_keypair.c`:

```
#include <oqs/oqs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    uint8_t* public_key;
    size_t public_key_len;
    uint8_t* private_key;
    size_t private_key_len;
} KeypairResult;

KeypairResult generate_PQ_keypair(int stampa) {
    KeypairResult result = {NULL, 0,NULL,0};

    if (!OQS_SIG_alg_is_enabled(OQS_SIG_alg_sphincs_shake_256s_simple)) {
        fprintf(stderr, "SPHINCS+ Algorithm not available!\n");
        return result;
    }

    OQS_SIG* sig = OQS_SIG_new(OQS_SIG_alg_sphincs_shake_256s_simple);
    if (sig == NULL) {
        fprintf(stderr, "Error in the creation of the signature
            structure!\n");
        return result;
    }
    result.public_key_len = sig->length_public_key;
    result.private_key_len = sig->length_secret_key;
```

```
    result.public_key = malloc(result.public_key_len);
    result.private_key = malloc(result.private_key_len);
    if (result.public_key == NULL || result.private_key == NULL) {
        fprintf(stderr, "Error in the allocation of the keys!\n");
        OQS_SIG_free(sig);
        free(result.public_key);
        free(result.private_key);
        return result;
    }

    if (OQS_SIG_keypair(sig, result.public_key, result.private_key) !=
        OQS_SUCCESS) {
        fprintf(stderr, "Error in the generation of the keys!\n");
        OQS_SIG_free(sig);
        free(result.public_key);
        free(result.private_key);
        return result;
    }


    if (stampa == 1){
    fprintf(stderr, "Public key: ");
    for (size_t i = 0; i < result.public_key_len; i++) {
        fprintf(stderr, "%02X", result.public_key[i]);
    }
    fprintf(stderr, "\n");

    fprintf(stderr, "Private key: ");
    for (size_t i = 0; i < result.private_key_len; i++) {
        fprintf(stderr, "%02X", result.private_key[i]);
    }
    fprintf(stderr, "\n");
    }

    OQS_SIG_free(sig);

    return result;
```

To use this file, we must build a shared library from the C file, install it in `/usr/local/lib/`, and register it with the system linker. It is possible by executing the following commands:

```
gcc -shared -fPIC gen_keypair.c -o libgen_keypair.so -I/usr/local/include
    -L/usr/local/lib -lcrypto -lssl -loqs
sudo cp libgen_keypair.so /usr/local/lib/
sudo ldconfig
```

B.2.3 Function for post-quantum signature generation

Content of the file `sign_with_PQ.c`:

```
#include <oqs/oqs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
```

```
    uint8_t* signature;
    size_t signature_len;
} SignatureResult;

SignatureResult sign_with_PQ(const uint8_t * quote, size_t quote_len, const
    uint8_t * pq_priv_key, size_t pq_priv_key_len){
    SignatureResult result = {NULL, 0};


    if (!OQS_SIG_alg_is_enabled(OQS_SIG_alg_sphincs_shake_256s_simple)) {
        fprintf(stderr, "SPHINCS+ algorithm not available!\n");
        return result;
    }


    OQS_SIG *sig = OQS_SIG_new(OQS_SIG_alg_sphincs_shake_256s_simple);
    if (sig == NULL) {
        fprintf(stderr, "Error in the creation of the structure!\n");
        return result;
    }


    result.signature = malloc(sig->length_signature);
    if (result.signature == NULL) {
        fprintf(stderr, "Error in the allocation of the signature!\n");
        OQS_SIG_free(sig);
        return result;
    }

    result.signature_len = sig->length_signature;

    if (OQS_SIG_sign(sig, result.signature, &sig->length_signature, quote,
        quote_len, pq_priv_key) != OQS_SUCCESS) {
        fprintf(stderr, "Error in the signature!\n");
        free(result.signature);
        OQS_SIG_free(sig);
        return result;
    }

    OQS_SIG_free(sig);
    return result;
}
```

After the creation of the file, run the following commands:

```
gcc -shared -fPIC sign_with_PQ.c -o libsign_with_PQ.so -I/usr/local/include
    -L/usr/local/lib -lcrypto -lssl -loqs
sudo cp libsign_with_PQ.so /usr/local/lib/
sudo ldconfig
```

B.3 Keylime Registrar modifications

In the file `registrar_common.py`add the verification of the signature received from the Agent during the challenge-response phase, needed to validate the correctness of the public post-quantum key:

```
auth_tag = json_body["auth_tag"]
auth_tag_len = json_body["auth_tag_len"]
sign_pq = json_body["sign_pq"]
sign_pq_len = json_body["sign_pq_len"]
pq_key = json_body["pq_key"]
pq_key_len = json_body["pq_key_len"]


# Crea i puntatori ctypes per signature e public_key
signature_ptr = (c_ubyte * sign_pq_len)(*sign_pq) # Crea un array
    di uint8_t per la firma
public_key_ptr = (c_ubyte * pq_key_len)(*pq_key)
auth_ptr = (c_ubyte * auth_tag_len)(*auth_tag)

result = lib.verify_signature(auth_ptr, c_size_t(auth_tag_len),
    signature_ptr, c_size_t(sign_pq_len), public_key_ptr)

if result == 0:
       logger.info("Verification of PQK: Valid")
else:
       raise Exception(
       f"Auth tag {auth_tag} for agent {agent_id} does not match
           expected value. The agent has been deleted from
           database, and a restart of it will be required"
    )
```

Then, add the post-quantum key field in the response body of the `do_GET` function:

```
response = {
        "aik_tpm": agent.aik_tpm,
        "ek_tpm": agent.ek_tpm,
        ...
        "pq_key": agent.pq_key,
    }
```

and add the value in the database within the `do_POST` function:

```
pq_key = json_body["pq_key"]

# Add values to database
d: Dict[str, Any] = {
    "agent_id": agent_id,
    "ek_tpm": ek_tpm,
    "aik_tpm": aik_tpm,
    "ekcert": ekcert,
    ...
    "pq_key": pq_key,
}
```

Extend the structure in the file `registrar_db.py`:

```
class RegistrarMain(Base):
__tablename__ = "registrarmain"
agent_id = Column(String(80), primary_key=True)
key = Column(String(45))
aik_tpm = Column(String(500))
...
pq_key = Column(String(500))
```

In file `registrar_agent.py` add the function `_validate_pq_key`:

```
def _validate_pq_key(self, pq_key: str) -> None:
    if not isinstance(pq_key, str):
       raise ValueError("pq_key must be a string")
    try:
    # Prova a decodificare Base64 per verificare il formato
      decoded_key = base64.b64decode(pq_key)
    except Exception:
      raise ValueError("Invalid pq_key: not a valid Base64 string")
```

In file `registrar_client.py` modify the `getData` function by adding the check on the Post-Quantum key:

```
 if "pq_key" not in response_body["results"]:
       logger.critical("Error: did not receive Post-Quantum key from
           Registrar Server.")
       return None
```

and the insertion in the `RegistrarData` class:

```
  res: RegistrarData = {
       "aik_tpm": r["aik_tpm"],
       "regcount": r["regcount"],
       ...
       "pq_key" : r["pq_key"],
   }
```

B.3.1 Modification of the Registrar DB

To modify the `registrarmain` table in the SQLite database to include the Post-Quantum key field, the following steps were performed: the original table is renamed to preserve its structure and data during the migration

```
ALTER TABLE registrarmain RENAME TO registrarmain_old;
```

A new table with the updated schema is created, adding `pq_key` as a `BLOB` to accommodate its binary nature:

```
CREATE TABLE registrarmain (
    agent_id VARCHAR(80) PRIMARY KEY,
    key VARCHAR(45),
    ekcert TEXT,
    virtual BOOLEAN,
    active BOOLEAN,
    provider_keys TEXT,
    regcount INTEGER,
    aik_tpm VARCHAR(500),
    ek_tpm VARCHAR(500),
    ip VARCHAR(15),
    port INTEGER,
    mtls_cert TEXT,
    iak_tpm VARCHAR(500),
    idevid_tpm VARCHAR(500),
    iak_cert VARCHAR(2048),
    idevid_cert VARCHAR(2048),
    pq_key BLOB
);
```

Data from the old table are transferred to the new table, ensuring existing data to be preserved:

```
INSERT INTO registrarmain (
    agent_id, key, ekcert, virtual, active, provider_keys, regcount,
    aik_tpm, ek_tpm, ip, port, mtls_cert, iak_tpm, idevid_tpm, iak_cert,
    idevid_cert, pq_key
)
SELECT
    agent_id, key, ekcert, virtual, active, provider_keys, regcount,
    aik_tpm, ek_tpm, ip, port, mtls_cert, iak_tpm, idevid_tpm, iak_cert,
    idevid_cert, pq_key
FROM registrarmain_old;
```

Finally, the old table is deleted to remove redundancy:

```
DROP TABLE registrarmain_old;
```

B.4 Keylime Verifier modifications

In the file `cloud_verifier_tornado.py` add the declaration of the signature verification function:

```
lib.verify_signature.argtypes = [
    POINTER(c_ubyte), # uint8_t* message
    c_size_t,
    POINTER(c_ubyte), # uint8_t* signature
    c_size_t, # size_t signature_len
    POINTER(c_ubyte) # uint8_t* public_key
]
lib.verify_signature.restype = ctypes.c_int
```

In the struct `exclude_db` add the field related to the post-quantum key:

```
exclude_db: Dict[str, Any] = {
    "registrar_data": "",
    "nonce": "",
    "b64_encrypted_V": "",
    ...
    "pq_key" : "",
}
```

In function `invoke_get_quote` add the code needed to call the handle the Post-quantum signature, specifically extracting the fields:

```
quote = json_response.get("results", {}).get("quote").encode('utf-8')
quote_len= json_response.get("results", {}).get("quote_len")
sign_PQ = bytearray(json_response.get("results", {}).get("sign_PQ"))
sign_PQ_len = json_response.get("results", {}).get("sign_PQ_len")
pq_key = bytearray(json_response.get("results", {}).get("pq_key"))
pq_key_len= json_response.get("results", {}).get("pq_key_len")
```

The conversion of the quote, the signature and the public key into python pointers:

```
signature_ptr = (c_ubyte * sign_PQ_len)(*sign_PQ) # Crea un array di uint8_t
    per la firma
public_key_ptr = (c_ubyte * pq_key_len)(*pq_key)
quote_ptr = (c_ubyte * quote_len)(*quote)
```

and then the part that handles the verification of the signature:

```
if sign_PQ is None or sign_PQ_len is None or pq_key is None:
          logger.warning("missing_fields", "One or more required fields not
              found in Agent's response.")
          # Gestisci l'errore come preferisci
          failure.add_event("missing_fields", "One or more required fields
              not found in Agent's response", False)
          asyncio.ensure_future(process_agent(agent, states.FAILED, failure))
          return

      if pq_key == pq_key_registrar:

          logger.info("PQ key received correctly \n")
          result = lib.verify_signature(quote_ptr, c_size_t(quote_len),
              signature_ptr, c_size_t(sign_PQ_len), public_key_ptr)


          if result == 0:
              logger.info("Verification of Sphincs signature: Valid")
              global counter
              counter = 0
          else:
              logger.error("Verification of Sphincs signature: Not valid")
              counter +=1
              if counter == 2 :
                  failure = Failure(Component.QUOTE_VALIDATION)
                  failure.add_event("invalid Sphincs signature",{"message":
                      "Sphincs Public Key is not corresponding to the correct
                      one"},False)
                  asyncio.ensure_future(process_agent(agent,
                      states.INVALID_QUOTE, failure))
      else:
          failure = Failure(Component.QUOTE_VALIDATION)
          failure.add_event("invalid Sphincs signature",{"message": "Sphincs
              Public Key is not corresponding to the correct one"},False)
          asyncio.ensure_future(process_agent(agent, states.INVALID_QUOTE,
              failure))
          logger.error("Sphincs Public Key is not corresponding to the
              correct one")
```

B.4.1 Function for verification of the signature

Content of the file `verifysign.c`:

```
#include <oqs/oqs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int verify_signature(const uint8_t* message, size_t quote_len, const uint8_t*
    signature, size_t signature_len, const uint8_t* public_key) {
    if (!OQS_SIG_alg_is_enabled(OQS_SIG_alg_sphincs_shake_256s_simple)) {
        fprintf(stderr, "SPHINCS+ algorithm not available!\n");
        return -1;
```

```
    }

    OQS_SIG *sig = OQS_SIG_new(OQS_SIG_alg_sphincs_shake_256s_simple);
    if (sig == NULL) {
        fprintf(stderr, "Error in the creation of the SPHINCS structure!\n");
        return -1;
    }
    if (OQS_SIG_verify(sig, message, quote_len, signature,
        sig->length_signature, public_key)!= OQS_SUCCESS) {

        return -1;
    }

    return 0;
}
```

After the creation of the file, run the following commands:

```
gcc -shared -o verifysign.so -fPIC verifysign.c -loqs -lcrypto -lssl
sudo cp verifysign.so /usr/local/lib
sudo ldconfig
```

B.5 Keylime tenant modifications

In the `tenant.py` file, add the field related to the post-quantum key in the `do_cvadd` function:

```
 data = {
        "v": b64_v,
        "cloudagent_ip": self.cv_cloudagent_ip,
        "cloudagent_port": self.agent_port,
        "verifier_ip": self.verifier_ip,
        "verifier_port": self.verifier_port,
        "tpm_policy": json.dumps(self.tpm_policy),
        "runtime_policy": self.runtime_policy,
        ...
        "pq_key": self.registrar_data["pq_key"],
    }
```