Post-Quantum Cryptography in the Messaging Layer Security Protocol

Severin Ullavik Erstad

Thesis for Master of Science Degree at the
University of Bergen, Norway

2025

Year: 2025
Title: Post-Quantum Cryptography in the Messaging Layer Security Protocol
Author: Severin Ullavik Erstad

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Håvard Raddum, for suggesting such an engaging thesis project and for providing continuous and invaluable support throughout this project.

To my partner, Lovise Kaspara. Thank you for your patience and care. You have been my greatest reminder of what truly matters.

I thank my family for their continuous support, for indulging in conversations with me about topics that are likely far from interesting to them, and for fostering my curiosity. You are all wonderful role models.

Special thanks go to my good friend Birk, whose company, conversations, and occasional rants offered much-needed diversions.

Abstract

The continuous development of quantum technologies poses a significant threat to traditional public-key cryptography, necessitating a transition to post-quantum algorithms capable of protecting against attacks from future cryptographically relevant quantum computers.

Messaging Layer Security (MLS) is a fairly new protocol that uses ratchet trees to provide scalability for end-to-end encrypted group messaging beyond that of traditional approaches. The scalability of MLS makes the transition to post-quantum cryptography feasible for larger groups. However, the current standard does not yet offer post-quantum security.

This thesis presents a detailed study of the MLS protocol, its cryptographic design, and the concepts that ensure its efficiency and security features. We also examine the ongoing efforts of the MLS Working Group to introduce post-quantum security to MLS.

Furthermore, we examine the impact in performance and message-size in transitioning to post-quantum key exchange mechanisms using OpenMLS, an open-source implementation of MLS that supports some post-quantum cipher suites.

Our analysis provides insights into the practical feasibility and overhead of post-quantum MLS.

Statement on the use of AI tools

In this thesis, OpenAI's LLM GPT-5 has been utilized as a support tool on several occasions.

GPT-5 has assisted in LaTeX programming, mainly providing templates and boilerplates for figures, equations, and references. All of these have been reviewed and verified by me.

In addition, GPT-5 has been used to clean sentences of errors related to structure, spelling, and grammar.

I am aware that I am responsible for all the content of this master's thesis.

Contents

Chapter 1

Introduction

Cryptography is the study of tools and mechanisms for secure communication such that certain favorable properties are guaranteed for the storage and transmission of sensitive data over insecure networks.

While cryptography is the science of securing data, cryptanalysis is the science of analyzing and breaking secure communication. Together, cryptography and cryptanalysis constitute the two fundamental pillars of cryptology. The main security properties cryptography aims to achieve are *confidentiality*, *integrity*, *authenticity*, and *non-repudiation*. These security goals can be defined as follows.

- Confidentiality: Information is kept secret from all but intended parties.
- Integrity: The message has not been modified after creation.
- Authentication: Assurance of the origin of the message.
- Non-repudiation: The sender of a message cannot deny ownership.

1.1 Cryptography

Cryptographic systems are constructed from various building blocks that operate at a lower level. Cryptographic *primitives* are fundamental algorithms designed to perform specific security tasks, such as ciphers, digital signatures, and hashes. Cryptographic *protocols* use these primitives to define structured procedures.

1.1.1 Cryptographic Goals

Applications weigh the core properties of cryptography differently according to their specific needs. For example, an online banking solution relies heavily on all of them. To protect privacy, only the owner of a bank account should be able to examine its financial information, ensuring **confidentiality**. All transactions must be recorded exactly as intended, preserving **integrity**. Before interacting with the account, the user must be identified as its legitimate owner, providing **authentication**. Finally, the owner must not be able to deny having requested the change, such as an unfortunate investment, thereby ensuring **non-repudiation**.

In practice, the highest level of security is not always ideal. It often correlates with more demanding computations and is usually not necessary. Instead, cryptographic systems should be balanced around trade-offs in risk and performance.

1.1.2 Cryptography in Communication

These properties are essential in digital communication. To ensure that communicating entities can correctly *encrypt* and *decrypt* to each other to successfully exclude any third parties, they must first possess one or more secrets that cannot be accessed or easily discovered by others. This secret is referred to as the *key*. Encryption uses the key to scramble messages, thus ensuring the confidentiality of its contents. The recipient's key, in turn, restores the message to its original state. Along with additional methods such as digital signatures to authenticate the sender and hash functions to verify the integrity of transmitted data, these enable secure communication across insecure networks.

1.2 Brief History of Cryptography

The practice of cryptography has evolved significantly over the centuries, shaped by the changing demands of secrecy and means of communication. The first known use of cryptography is found in hieroglyphs from the Old Kingdom of Egypt (EST 1900 BC) [17] almost 4000 years ago. Approximately 1700 years later, encryption is mentioned as one of the arts for a better quality of life in the Indian-written Kama Sutra. The method suggested is

to pair the letters of the alphabet and substitute each letter in a text with its partner letter [16].

1.2.1 Classical Ciphers

Classical techniques such as the Caesar cipher (EST 100-50 BC), a letter substitution cipher supposedly used by Julius Caesar, and the Scytale (EST 50–120 AD), a transposition method employed by the Greeks, were designed to obscure plaintext from unintentional readers. These classical ciphers offered confidentiality, but relied heavily on the secrecy of the method rather than the strength of the algorithm.

The method of the Caesar cipher would simply shift every letter in the plaintext by a predetermined number of positions in the alphabet, according to the key. Once a key was agreed upon, for example, 3, the sender would shift each letter by three positions. 'A' encrypts to 'D', 'B' encrypts to 'E', and so on. Note that the alphabet wraps around so that, e.g., 'X' encrypts to 'A'. The recipient would retrieve the original message by applying the reverse process to the ciphertext.

One of its major weaknesses is that the set of possible keys, the *key space*, is limited by the length of the alphabet. Because of this, it is feasible for an adversary to break the cipher by iterating each key in the key space until the ciphertext decrypts to a meaningful message, known as a *brute-force attack*.

During the Islamic Golden Age (8-13. Century AD), Scholar Al-Kindi introduced the practice of breaking cryptographic systems by identifying and exploiting the letter frequency of the text language to break ciphers, a method known as *frequency analysis* [16]. By matching the frequent letters in the ciphertext with frequent letters in the language, one greatly increases the success rate of extracting information about the key.

Against the Ceasar cipher, one could reasonably assume that the key used to encrypt corresponds to the distance between the most common letter in the plaintext and the most common letter in the ciphertext. As the letter frequencies in messages sometimes deviate some from that of the language, some guesses might be necessary.

In response, cryptography had to adapt to advances in cryptanalysis. To combat the one-to-one correspondence between letters in the plaintext and ciphertext, thus making frequency analysis more difficult, more advanced ciphers were developed. In the sixteenth century, the Vigenère cipher was

introduced. It is essentially a polyalphabetic variation of the Ceasar cipher, where the key is larger, thus producing less predictable ciphertexts. When encrypting a message, the key is repeated for the entire plaintext. For example, when choosing key [1, 6, 3, 2], the first letter would shift 1 position, the second 6 positions, then 3 positions, 2 positions and 1 position again.

Although the cipher produced a more complex relationship between plaintext and ciphertext, its reuse of the key proved to expose vulnerabilities. Cryptanalysts discovered that certain patterns in the plaintext could be pinpointed in the ciphertext. For instance, 'the' is a recurring word in most English texts. Cryptanalysts could search for a similar recurring pattern in the ciphertext and then use that information to deduce the length of the key. Once the length of the key is known, frequency analysis could be applied to recover the key and plaintext.

1.2.2 The World War II

In the twentieth century, especially during the World Wars, cryptography entered a new era in scale and complexity. The use of mechanical cipher machines, such as the German Enigma, introduced dynamic substitution systems, where a different substitution alphabet would be applied to encrypt each letter in the plaintext.

The communicating parties would configure their Enigma machines in the same way, this configuration serving as their secret key. The subsequent communication would then be encrypted and decrypted when processed by their respective machine.

Although the Enigma Machine was considered secure enough to encrypt much of the correspondence of the Axis powers during WWII [10], comprehensive Allied efforts to break it finally succeeded, largely due to Poland sharing their cryptanalytic achievements [8].

1.3 Modern Cryptography

Modern cryptography as we know it today began in the 1970s [52]. In 1977, the Data Encryption Standard (DES) was adopted as a Federal Information Processing Standard (FIPS) by the National Institute of Standards and Technology (NIST) [32].

A year earlier, Whitfield Diffie and Martin Hellman addressed the then open problem of secure key distribution using an asymmetric technique known as the Diffie-Hellman Key Exchange (DHKE)[11].

The Rivest-Shamir-Adleman (RSA) algorithm, also published in 1977, introduced asymmetric techniques for encryption and digital signatures, thus enabling confidentiality and authenticity without necessitating pre-shared symmetric keys.

Hash functions such as the Message-Digest algorithm (MD5) and the Secure Hash Algorithm 1 (SHA-1) provided mechanisms to verify the integrity of transmitted data.

In the late 1990s and early 2000s, the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols could be used to secure web traffic, and Pretty Good Privacy (PGP) to secure email traffic.

The Advanced Encryption Standard (AES) [34] replaced DES in 2001, providing better security against brute-force attacks, in which DES had become vulnerable due to its insufficient key size against increasing computational capacities.

Before the 1970s, cryptography was mainly used for military, government, and diplomatic purposes [52]. Today, cryptography has become part of our modern lives. We use it daily in tasks such as accessing our digital accounts and devices, messaging and calling each other, paying bills and online shopping, and many more.

1.3.1 Emerging Challenges

In recent years, cryptography has continued to evolve to meet emerging challenges. Advances in technology lead to faster computers and more attacks becoming feasible, which must be reciprocated with implementing stronger security. Modern times involve more internet traffic than ever before, and we expect our tasks to perform with minimal latency. At the same time, the threat of quantum computers necessitates *post-quantum cryptography*.

We currently rely on the practical difficulty of certain mathematical problems in our cryptosystems. The previously mentioned DHKE protocol and RSA algorithm rely on the hardness of the discrete logarithm problem and the prime factorization problem, respectively. The obstacle is that both of these problems, which are currently considered safe, are expected to be feasible to break for quantum computers.

Post-quantum cryptography (PQC) aims to address this concern. By adapting to algorithms that rely on the hardness of problems that quantum computing is not expected to feasibly break, PQC is anticipated to defend against quantum attacks, but has the drawback of increased computational costs.

Quantum computing is still a fairly new threat, and it is not expected that there will be cryptographically relevant quantum computers (CRQC) ready to attack our current systems in the next few years. At the same time, the existence of the Harvest Now, Decrypt Later (HDNL) attack, in which hackers are storing intercepted messages in anticipation of future computing capacities, emphasizes the urgency in transitioning to PQC.

Vulnerabilities are often discovered in cryptosystems that have not yet stood the test of time and scrutiny. Before replacing old standards with new ones, they must first be thoroughly tested and analyzed. NIST has already embarked on the project of standardizing post-quantum algorithms.

1.3.2 Secure Messaging

Cryptography is the tool we rely on for secure and reliable communication. The information we store and exchange must be protected against unauthorized access and modification. In 2013, Edward Snowden revealed that the US government monitored digital communications on a massive scale. This revelation highlighted the importance of cryptography in protecting privacy against institutional surveillance, not just malicious hackers. Since then, there has been greater emphasis on end-to-end encryption to ensure that messages remain confidential.

Modern messaging services end-to-end encrypt transmitted data to provide these safeguards. By transmitting encrypted data, interception does not pose an immediate threat to the confidentiality of its contents.

In Signal, a prominent protocol in modern communication, secrets are agreed upon and updated using the Double Ratchet algorithm. The algorithm continuously progresses the key material as messages are sent. In this way, the damage of a compromised key is limited, and the security of the rest of the communication remains intact.

This continuous update and distribution of key material works great for two communicating entities, but has substantial drawbacks in terms of scaling for larger groups, as updated key material must be distributed to

every other member in the group.

1.4 Problem Statement and Objectives

The Messaging Layer Security protocol (MLS) was made internet standard with RFC 9420 [2] in 2023. The protocol provides its users with continuous group authenticated key exchange while maintaining strong security and efficient scaling. However, the current standard for MLS does not offer post-quantum security.

The objective of this thesis is to study the specification in RFC 9420 and to present how MLS achieves its goals. Firstly, we explore how MLS utilizes a ratchet tree structure to reduce the number of required operations to scale with the logarithm of the group size, as opposed to linear scaling in traditional approaches. Secondly, we look at how the cryptographic design of MLS achieves its security features.

In addition, we will examine the impact on performance and memory for MLS groups when replacing traditional algorithms with post-quantum ones. Several libraries implementing MLS are under development, and some support post-quantum algorithms. We will use the OpenMLS library to simulate classical and post-quantum MLS sessions for various group sizes and report on the behavior. In particular, inspecting how MLS responds to computationally demanding burden of PQC. In comparing traditional and post-quantum MLS sessions, our objective is to provide some insight into the cost of upgrading to PQC in the protocol.

Chapter 2

Preliminaries

In this section, we focus on the various cryptographic methods used to achieve the core cryptographic properties in modern communication. In addition, some topics relevant to understanding the MLS protocol are covered.

2.1 Symmetric Key Cryptography

In symmetric cryptography, the same key is used by both sides of the communication. As shown in Figure 2.1, the key used to encrypt the plaintext is equivalent to the key used to decrypt the ciphertext. Applying any other key to decrypt would result in a meaningless output. In this way, symmetric ciphers ensure that the content of messages remains confidential between the communicating parties, given that they use secure ciphers and key establishment methods.

Symmetric cryptography consists mainly of various *stream and block ciphers*. Stream ciphers encrypt plaintext one logical bit at a time. The shared key is extended to the size of the plaintext using a *keystream generation function* that must be difficult for an adversary to replicate. The generated keystream is then applied to the plaintext to compute the ciphertext. The recipient will produce the same keystream from the key and use it to decrypt the message. ChaCha20 is an example of a widely used and secure stream cipher.

Block ciphers instead encrypt the message block by block. To encrypt, the plaintext is split into fixed-size blocks which are encrypted one block at a time. This allows for a more complex relationship between plaintext

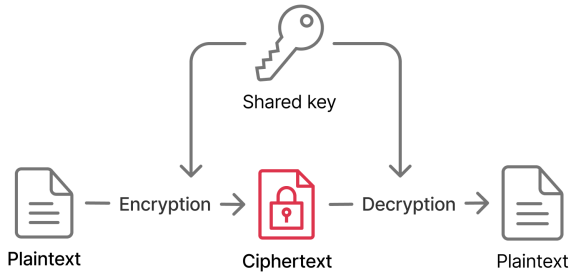

Figure 2.1: Symmetric encryption and decryption

and ciphertext, as results from encrypted blocks during the process can be used to affect the encryption of later blocks. Methods for doing this are called *modes of operations*. Confusion and diffusion are important principles in the construction of secure block ciphers. Strong confusion implies a complex relationship between the key, the plaintext, and the ciphertext, making it harder for cryptanalysis to recover the key or the plaintext from the ciphertext. Strong diffusion implies that any small change in plaintext or key should result in a great alteration in the ciphertext they produce. Feistel- and substitution-permutation networks are trusted frameworks for designing block ciphers that take advantage of these principles. The Advanced Encryption Standard (AES) [33] is a prominent symmetric encryption algorithm that builds its security on a substation-permutation network.

2.1.1 Cryptographic Hash Functions

Hash functions have broad applications in computer science. They take an arbitrary-length input and produce a fixed-size output called a hash digest. To qualify as a cryptographically secure hash function, the functions must possess the following properties [40]:

- Collision resistance: It is infeasible to find two inputs that produce the same output

- Preimage resistance: It is infeasible to compute the input from the digest
- Second preimage resistance: Given an input and its digest, it is infeasible to find another input that produces the same digest.

Hash functions can be constructed similarly to block ciphers using modes of operations where the encryption of each block affects the encryption of the final block [19]. Then, the last encrypted block is the hash digest of the entire input.

Hash functions are often used to increase efficiency in other computationally demanding schemes where the size of the input matters. They are often used in conjunction with digital signatures for this purpose, but they also have other uses in cryptography. The current standards for hash functions are SHA-2 and SHA-3 [39].

2.1.2 Message Authentication Codes and Authenticated Encryption with Associated Data

A message authentication code (MAC) is a small block of data sent along with the message. The shared key is used along with the message to compute an MAC.

MACs can be constructed from block ciphers or hash functions. In the latter case, they should be constructed as an HMAC, a hash-based MAC that inherits the security of the underlying cryptographic hash function used [20].

MACs are used to provide integrity and authentication in communication. When Bob receives a message and its MAC, as visualized in Figure 2.2, he computes his own MAC using the shared key and the message received. He can then verify it by checking that the transmitted MAC matches the MAC he computed. This assures him that the message has not been altered and that it was computed by someone who knows the shared key.

Similarly to MACs, Authenticated Encryption with Associated Data (AEAD) transmits some unencrypted associated data (AD) along with the ciphertext. However, AEADs are a type of encryption that in addition to confidentiality provides integrity and authentication, thus combining the security benefits of ciphers and MACs [26].

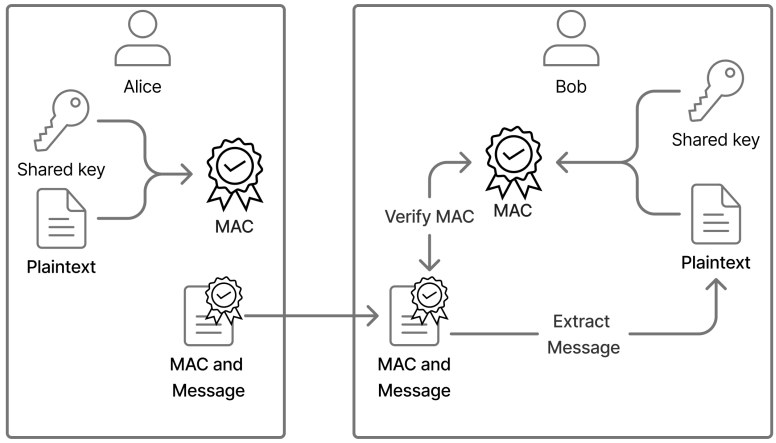

Figure 2.2: Message Authentication Codes

2.2 Asymmetric Key Cryptography

So far, we have considered symmetric methods that use the same shared key for encryption and decryption. However, in asymmetric cryptography, separate keys are used to encrypt and decrypt.

In asymmetric cryptography, there are two keys related to each participant. One *public key* which can be distributed openly and one *private key* which is kept secret.

The relationship between the two keys is based on complex computation problems from number theory which are considered difficult to solve. This is what allows the public key to be publicly shared without affecting the security of the private key. Although the problems are hard to solve, the two asymmetric keys effectively function as *trapdoors* to easily revert the computation of the other.

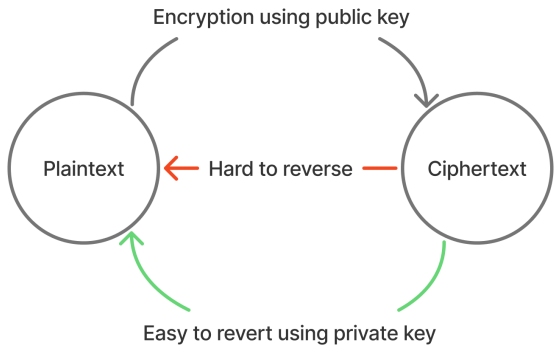

Figure 2.3: One-way function with trapdoor

Figure 2.3 shows this relationship when a public key is used to encrypt. The private key has the necessary information to reverse the encryption and regain the plaintext, a process which should be infeasible without the trapdoor. Examples of hard computational problems used in asymmetric cryptography are the discrete logarithm and integer factorization problems.

2.2.1 Asymmetric Ciphers

When Alice wants to message Bob, she retrieves Bob's openly distributed public key and uses it to encrypt the message. Due to the intricate relationship between Bob's keys, only his private key can be used to successfully decrypt the message. This is visualized in Figure 2.4. If Bob decides to respond to Alice, he in turn encrypts using Alice's public key.

The RSA cryptosystem was introduced in the 1970s and is still considered secure in modern times. It can be used to generate asymmetric key pairs and for encryption and decryption of data. RSA relies on the complexity of the integer factorization problem. To generate an asymmetric key pair using RSA, Bob proceeds with the following steps:

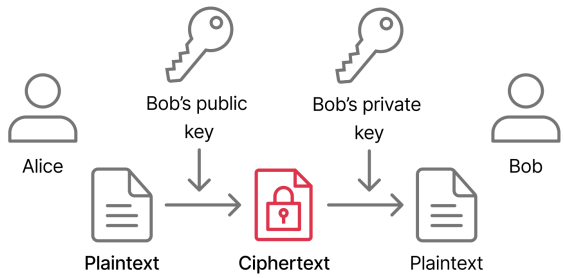

Figure 2.4: Asymmetric encryption and decryption

1. Select two large primes $p, q, \quad p \neq q$
2. Compute the modulus n :
   $$n = p \cdot q.$$
3. Compute m :
   $$m = (p-1)(q-1).$$
4. Choose the public exponent e such that
   $$1 < e < m \quad \text{and} \quad \gcd(e, m) = 1.$$
5. Compute d, the modular inverse of e :
   $$d = e^{-1} \pmod{m}.$$

From these calculations, Bob can start communicating securely using d as his private key and openly sharing (n, e) as his public key.

An attacker must recover d to get Bob's private key. However, to find d, the attacker must first discover p and q. We assume that any third party has access to Bob's public key, which contains (n, e). Extracting p and q from Bob's public key is an example of the integer factorization problem. In other words, there is no known solution to efficiently find p and q.

Every communicating participant using RSA must generate its key pairs

using the same procedure. Once the keys are generated, the communication using RSA follows the structure of Figure 2.4 using the following computations for encryption and decryption. Given a message x and its corresponding ciphertext y:

$$y = x^e \pmod{n},$$
$$x = y^d \pmod{n}.$$

However, RSA requires large values for p and q for its security, resulting in expensive computations for modular exponentiation. Frequent encryptions and decryptions of large messages are therefore inefficient.

Because of this, they are often used in conjunction with symmetric schemes. *Key encapsulation mechanisms* (KEM) combine them by using an asymmetric cipher for encryption of a symmetric key to establish a shared secret key used in subsequent communication.

2.2.2 Key Establishment

Asymmetric principles provide a secure and efficient way to establish shared keys for symmetric communication. The key material used in establishment can be transmitted in plaintext without compromising the security of its produced key. A prominent protocol for this is the Diffie-Hellman Key Exchange (DHKE).

For Alice and Bob to establish a shared secret key using DHKE, they start by selecting and openly publishing two numbers p and g, where p is a large prime and g must be a generator of the cyclic group $\mathbb{Z}_p$. Alice and Bob then choose a random number, a and b, respectively, within the realm of $\mathbb{Z}_p^*$. Proceeding, they compute and transmit g to the power of their respective number: Alice computes $A = g^a \pmod{p}$ and Bob computes $B = g^b \pmod{p}$. Once receiving B from Bob, Alice can compute the shared key $K = B^a \pmod{p}$. Bob reaches the same shared key by computing $K = A^b \pmod{p}$.

Since $K = A^b = B^a = g^{ab} \pmod{p}$, Alice and Bob have reached a shared key using DHKE that can be securely used for efficient communication using symmetric ciphers. DHKE relies on the hardness of the discrete logarithm problem, in which there is no known solution to efficiently compute a from $A = g^a \pmod{p}$.

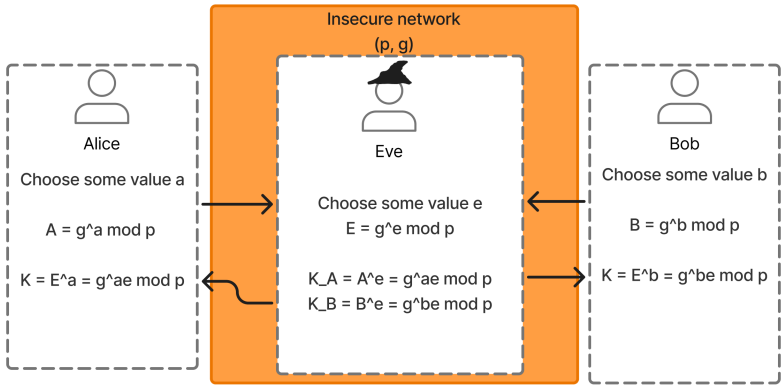

Figure 2.5: Man in the middle attack against DHKE

However, DHKE is vulnerable to *man in the middle* attacks. As shown in Figure 2.5. some third party, Eve, might intercept key material during establishment and forward false key material. This results in Alice and Bob establishing keys with Eve instead of each other. Alice and Bob will think they have established a secure key, while in reality all their communication goes through Eve, who can read, alter, and forward any messages at will.

2.2.3 Digital Signatures

Digital signatures provide assurances that you are in fact communicating with the intended recipient. Instead of signing entire messages, which would add a notable computational cost, one typically signs the hash digest of the message.

Digital signatures utilize the key pair in reverse order to the asymmetric ciphers we have mentioned. The message is signed using the private key of the sender and verified by the recipient using the public key of the sender.

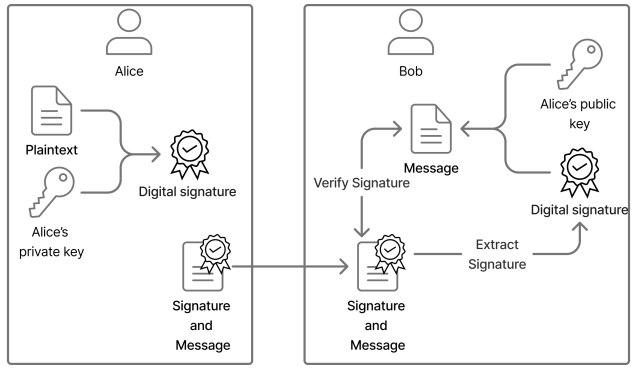

Figure 2.6: Signing a message and verifying a signature

Alice uses her private key to sign the message and transmits the signature along with the message. Bob receives the pair and applies the reverse operation to the signature using Alice's public key to compute the message as shown in Figure 2.6. The signature is verified if and only if the computed message matches the one received along with the signature. By verifying the signature, Bob confirms that it was produced using Alice's private key and is therefore assured of her authenticity. In addition, since the signature is affected by the message, Bob is assured of its integrity. Lastly, non-repudiation is provided, as Alice cannot deny having signed the message being the only entity in possession of her private key. The Digital Signature Algorithm (DSA), standardized in FIPS-186, is one of the secure standards for digital signatures.

2.3 Secure and reliable communication

Asymmetric cryptography solves the long-standing problem of shared key distribution. However, it has drawbacks in terms of efficiency, as they tend to perform slower than symmetric schemes. They are therefore commonly used in combination in cryptographic systems. For example, asymmetric schemes can be used to establish keys and add additional security benefits from digital signatures, whereas symmetric schemes are used to efficiently encrypt large sets of data.

By utilizing the strengths and properties offered by these symmetric and asymmetric schemes in various combinations, they are used to build secure and reliable cryptosystems and services.

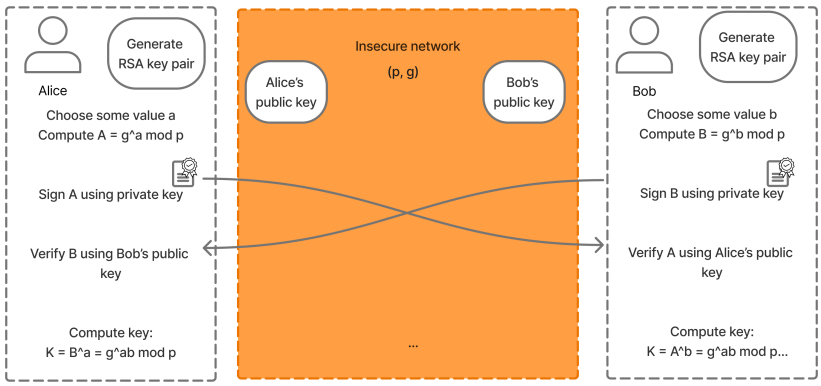

Figure 2.7: Combined RSA and DHKE

For example, to protect DHKE against impersonation attacks such as Man in the Middle, digital signatures can be applied to the transmitted key material as in Figure 2.7. By doing so, users can be sure of the integrity and authenticity of the established key.

Figure 2.8 displays an example of a cryptographic system in which a hash function, a digital signature, and a symmetric cipher work together to ensure the integrity and confidentiality of the message, as well as the authenticity of the sender.

2.3.1 End-to-end encryption

In real-world applications, the devices used in communication are not directly connected. Instead, they are both connected to some server or service consisting of several servers. Transmitted messages travel through this midpoint, which relays the message from sender to recipient. This could cause vulnerabilities if not handled correctly, as an attacker could intercept the message on its path from the sender to the server. Additionally, it creates a

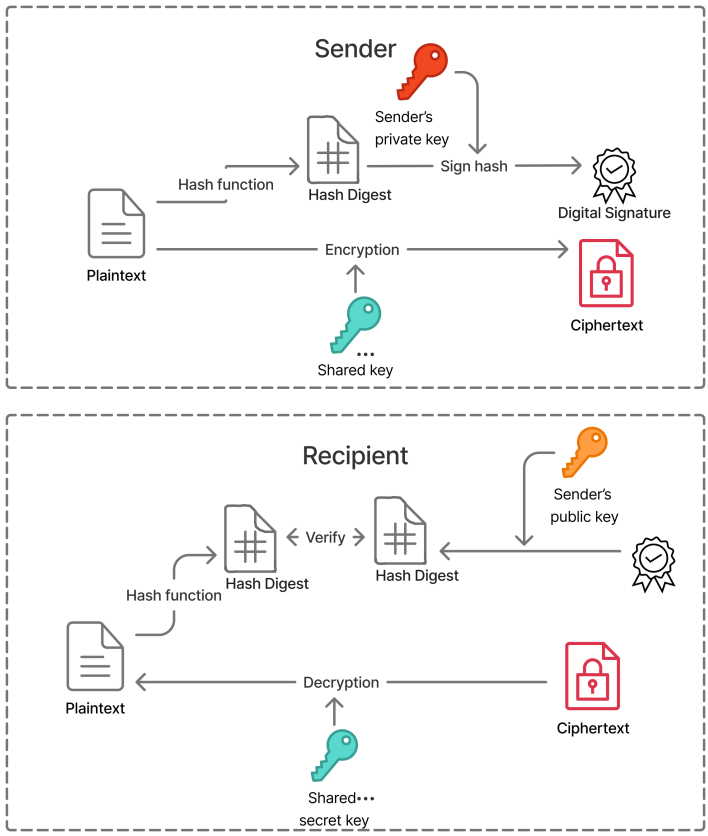

Figure 2.8: Example of a cryptographic system

high risk at the server, as an exploit could result in the loss of confidentiality for all its users, not just Alice and Bob.

As the server does not need to know the contents of the message to relay it, the solution is *end-to-end encryption* (E2EE). In E2EE, data encryption and decryption occur at the end points, that is, the devices used by the parties communicating. This results in all intermediaries, including the server and any adversaries, being unable to read the contents of the messages sent.

2.3.2 Additional Cryptosystem Properties

For continuous communication, some additional properties are favorable. For proper security, the keys used should be updated regularly. *Asynchronous key updating* is a property in which these updates can occur asynchronously without necessitating all communicating parties to be present or online.

Regular key updates protect the security of past and future communication in the event of a compromised key. For example, if the key is updated for every message sent, a compromised key should result only in loss of confidentiality for the single message it was used to encrypt.

Forward secrecy (FS) is a property that ensures that compromised keys cannot be used to obtain previous keys. This ensures that even if an attacker gains access to one of the keys of the communication, the security of all previous messages encrypted with earlier keys remains intact.

Post-compromise security (PCS) ensures that a compromised key cannot be used to derive any updated keys, thereby ensuring that an attacker with access to a compromised key will be excluded again when keys are updated.

2.3.3 The Double Ratchet Algorithm

Modern messaging applications, such as Signal, Whats-App, and Facebook Messenger, use the double ratchet algorithm in their end-to-end encrypted services.

The double ratchet algorithm, originally introduced with the Signal protocol [29] uses a ratchet function. The ratchet is used to efficiently compute new output that is hard to reverse, based on a variant of the previously explained hash functions. As a mechanical ratchet, its one-way property allows it to be clicked in one direction to derive new output but not in the other to retrieve old ones. As shown in Figure 2.9, the ratchet is used to derive keys and is therefore aptly referred to as a key derivation function ratchet.

The ratchet continuously updates the keys used in communication, and its one-way property provides forward secrecy. To also obtain post-compromise security, an attacker must be excluded from deriving updated keys even if he has access to the ratchet and a compromised key. This is achieved in the double ratchet algorithm by utilizing DHKE. By including secrets established using DHKE to derive new keys with the ratchet, an attacker will be automatically expelled soon after the breach.

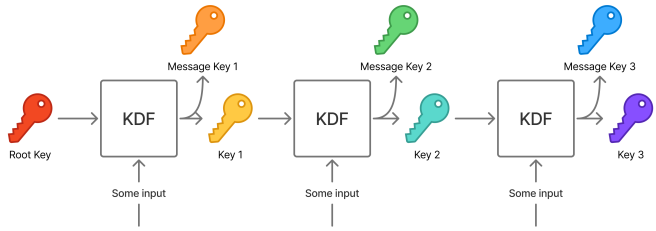

Figure 2.9: Key Derivation Function Ratchet

Both participants in the communication maintain a sender ratchet and a receiver ratchet. In the example of Alice and Bob, Alice's sender ratchet would be synchronized with Bob's receiver ratchet, and vice versa. When Alice sends a message, she uses the output key from clicking her sender ratchet. Bob can decrypt using the key produced from his receiver ratchet. This additionally provides asynchronous key updates. Multiple key updates can occur from several sender ratchet clicks, and the receiver will synchronize to this by performing an equivalent number of clicks to his receiving ratchet.

2.3.4 Group Messaging

Until now, our considerations have mostly been restricted to the communication of two parties, but a great deal of correspondence occurs in groups with more than two members. Group messaging has the same security goals: Confidentiality, integrity, authenticity, non-repudiation, as well as forward secrecy and post-compromise security. In addition, asynchronous key updates have a higher priority. Alice and Bob should be able to continue their communication and update their keys correspondingly, even when a third group member, Charlie, is out of reach. Charlie should also be able to read the messages sent to the group in his absence and synchronize with the key updates that occurred when he eventually comes online.

In a naïve approach, group messaging functions in the same way as two-party communication, but with some additional steps. Sending a message to the group corresponds to sending the same message to each of the group members individually. Thus, keys must be maintained and updated for each

co-member.

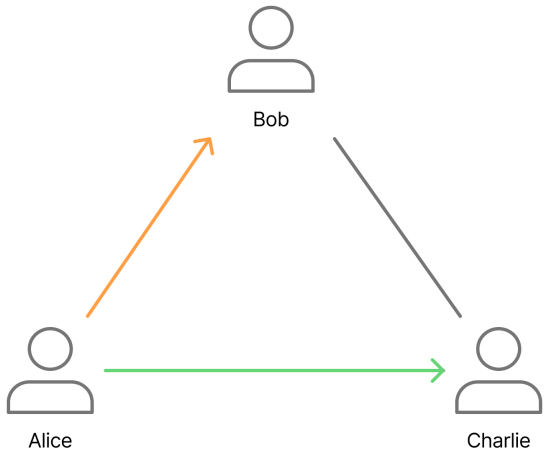

Figure 2.10: Encryption in a group with three members

This is visualized in Figure 2.10, where Alice encrypts to Bob and Charlie using the corresponding keys associated with each of them.

This approach causes the computational cost to grow linearly with the number of members in the group. $N-1$ operations, such as encryptions and individual key updates, are required in groups of N participants. In Figure 2.10, Alice encrypts the same message $3-1=2$ times. Linear scaling causes significant computational costs for large groups.

2.3.5 Messaging Layer Security for group messaging

The Messaging Layer Security (MLS) [2] protocol aims to solve the scalability issue in group messaging while efficiently maintaining strong security properties. By organizing group members as leaves and distributing keys to nodes in a *ratchet tree* structure, MLS effectively reduces the computational cost to scale with the logarithm of the group size. Thus, messaging and updating keys to the entire group only require $log_2(N)$ encryptions, where N is the number of members in the group.

This leads to a substantial save on computational resources for large groups. The reduction is especially beneficial as group messaging services over time must adapt to the threat of quantum computing. PQC require larger keys, signatures and ciphertexts, that cause an increased computational cost compared to traditional cryptography.

2.3.6 Tree Data Structure

Trees are a diverse and widely used data structure. They organize a set of data elements called *nodes* in hierarchical relationships. The nodes are connected vertically with *edges*. MLS ratchet trees are a specific type of tree data structure. To better understand the MLS protocol in chapter 4, some general terminology for trees are helpful. These are also visualized in Figure 2.11.

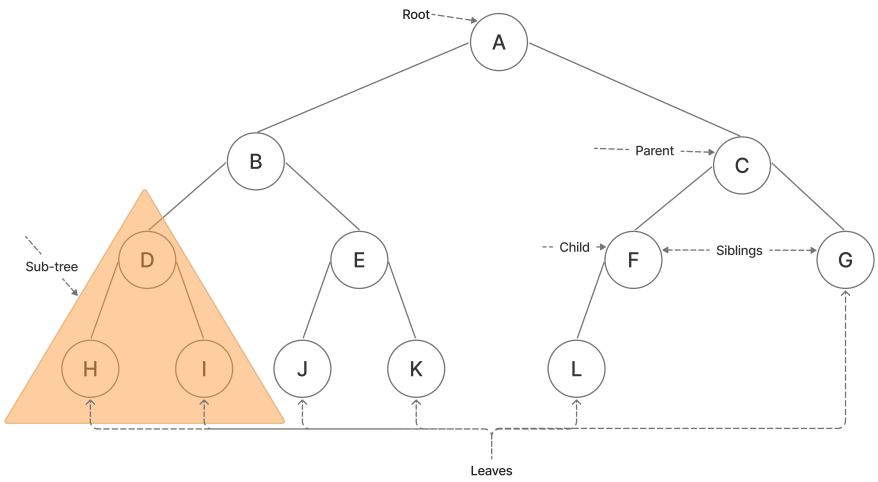

Figure 2.11: Binary tree data structure

The topmost node is known as the *root* node. For a given node in a tree, the node directly above is its *parent* if they are connected with an edge. The root node does not have a parent. Any nodes connected from below are referred to as the *children* of the parent node. A node is a *leaf* if it has no children. Any node that is neither the root nor a leaf is an *intermediate* node. The *descendants* of a node are the collection of its children and

all subsequent generations of children. Equivalently, the parent and all subsequent generations of parents are its *ancestors*. Nodes that have the same parent are *siblings*. A *subtree* is an extraction of the full tree consisting of a *head* node and its descendants. The three leftmost nodes in Figure 2.11 make up the subtree with node D as its head. The *depth* of a node is the number of edges that one must traverse through ancestor nodes to reach the root. This route from a node to the root is referred to as the *direct path* of the node. Siblings share the same depth level, and the root node is always at depth 0. The depth of the tree corresponds to the highest depth among its leaves. The ratchet trees in MLS are all perfect binary trees. In perfect binary trees, all parent nodes have exactly two children, and all leaves are at the same depth.

Chapter 3

Quantum Secure Cryptosystems

Modern cryptography relies on the computational intractability of solving certain mathematical problems, resulting in guarantees that the secret key and data remain hidden to all but intended parties.

In the 1980s, however, scientists began researching the idea of a new quantum computer system that utilizes quantum mechanics in computation [25]. These are expected to perform certain tasks significantly faster, and among these tasks are discrete logarithmic and integer factorization problems used in asymmetric cryptography.

3.1 Quantum Threat

Whereas classical computers process information as binary bits, 0's and 1's, quantum computers process quantum bits (qubits) which can exist in superposition as both at the same time [7]. These qubits can be entangled, a phenomenon in which two or more of them are linked so that the measurement of one affects the outcome of the other [31]. By utilizing these intricacies in quantum computers, they are expected to outperform classical computers to the point of feasibility in solving some hard problems [44].

3.1.1 Grover's and Shor's Algorithm

Grover's algorithm is a quantum search algorithm introduced by Lov K. Grover. Using the properties of quantum systems, it performs several operations simultaneously to identify a single record in an unsorted database [22].

The algorithm provides a quadratic speedup in comparison to any known classical search algorithm, reducing the time complexity from $T_{\text{classical}}$ to $\sqrt{T_{\text{classical}}}$. In cryptology, this is applied in brute-force attacks against symmetric schemes to reduce the attack time against some of our current standards from intractable to feasible.

Shor's algorithm, developed by Peter Shor in 1994, utilizes the strengths in quantum machines to efficiently solve both the discrete logarithm and the integer factorization problem [55]. This renders the security of asymmetric schemes such as RSA and DHKE broken.

In summary, quantum algorithms are expected to break the security of several of our widely used cryptographic standards. Although Grover's algorithm in theory speeds up brute-force attacks against symmetric schemes, it has later been shown that this speedup is impossible [6]. While this somewhat limits the quantum threat for symmetric schemes, Shor's algorithm necessitates a transition to *post-quantum cryptography* in place of the asymmetric standards used today.

Quantum computers capable of running these algorithms are referred to as *cryptographically relevant quantum computers* (CRQC). Currently, there are no practical CRQCs, but they are assumed to be possible [45]. It is hard to say when they will be ready, but they are anticipated to exist within the next few decades.

3.2 Post-Quantum Cryptography

When the time comes that the first CRQCs are introduced and we can no longer trust the security of modern cryptography, we need to already have adapted. The solution is *post-quantum cryptography* (PQC). PQC intends to replace vulnerable asymmetric schemes. Its security relies on computationally hard problems which are expected to be infeasible to solve for both classical and quantum computers.

Unfortunately, PQC algorithms are often significantly more computationally expensive due to larger keys and outputs, and may require adjusting existing and widely-used internet protocols for interoperability [6]. The five most prominent problems for this are *lattice-based*, *code-based*, *hash-based*, *multivariate-based*, and *isogeny-based cryptography*.

3.2.1 Standardization of PQC

Organizations and governments are beginning to prepare for the introduction of CRQCs by planning and migrating to PQC. For example, the US government aims to protect national security systems by completing the PQC migration by 2035 and provides assistance to achieve this goal[43]. In 2016, the National Institute for Standards and Technology (NIST) started an effort to standardize post-quantum alternatives for key establishment and digital signatures. In the format of an open competition, cryptographers could submit their algorithms for consideration. A total of 69 submissions met the minimum criteria [41].

Through rounds of evaluation and scrutiny over several years, NIST eventually standardized five post-quantum algorithms. They initially selected CRYSTALS-Kyber as the sole key encapsulation mechanism (KEM) standard, as well as CRYSTALS-Dilitium, SPHINCS+ and FALCON for digital signatures. A final round of the project was held to further consider candidates, which resulted in the choice of the Hamming Quasi-Cyclic (HQC) as an additional KEM standard [42].

Each of the PQC standards specifies multiple security parameters that offer different trade-offs in security and performance. There are a total of 5 security strength categories in order of increasing security strength [46]. These categories are defined according to the computational complexity of breaking classical algorithms. Strength level 1 must require computational resources comparable to or greater than those required for a key search on AES-128. The highest level, 5, is granted to configurations that demand comparable or greater computational resources than AES-256 [46].

3.2.2 Quantum-Safe Problems

The problems on which the standardized PQC algorithms build their security are lattice-, hash-, and code-based. Lattice-based cryptography uses lattices and the relative positions of points in them. It commonly derives its security from the shortest-vector problem and learning with errors problem. Hash-based cryptography already has a fundamental role in cryptography and has now proven to be useful in PQC signature schemes. Code-based cryptography builds its security on error-correction codes. The McEliece code-based cryptosystem has remained secure since its introduction in 1978

[6]

3.2.3 Standardized Post-Quantum Algorithms

The Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM) was the first key-encapsulation mechanism (KEM) standardized by NIST. Along with the standardized signature scheme Module-Lattice-Based Digital Signature Standard (ML-DSA), they both use lattice-based cryptography and build their security on the Module Learning with Errors problem [36] [35].

The Stateless Hash-Based Digital Signature Standard (SLH-DSA) is a hash-based digital signature scheme that is constructed using other hash-based schemes and takes advantage of their combined properties to securely sign messages [37].

The Fast Fourier Transform over NTRU-Lattice-Based Digital Signature Standard (FN-DSA) is another lattice-based signature standard. Although it has advantages over ML-DSA in terms of key- and signature size, it is considered difficult to understand and securely implement. Its official standard specification has not yet been published by NIST [54] [53].

The fourth round of the project selected Hamming Quasi-Cyclic (HQC) as an additional standard for KEM. HQC is code-based and builds its security on error-correcting codes. HQC requires larger keys and thus more demanding computational resources as opposed to ML-KEM, and therefore serves as a backup standard in case ML-KEM is discovered to be vulnerable [38].

3.3 PQC Migration

M.Mosca[28] highlights the urgency of transitioning to PQC using three variables, *security shelf-life*, *migration time*, and *collapse time*.

The security shelf-life denotes the time that a cryptographic scheme needs to provide confidentiality. This can vary substantially; while some data quickly lose their value and become deprecated, others are intended to remain private for much longer.

Migration time correlates to how long it will take to properly transition to post-quantum schemes. Previous transitions have taken close to twenty

years [45].

The final variable, the collapse time, is the time until a CRQC is introduced.

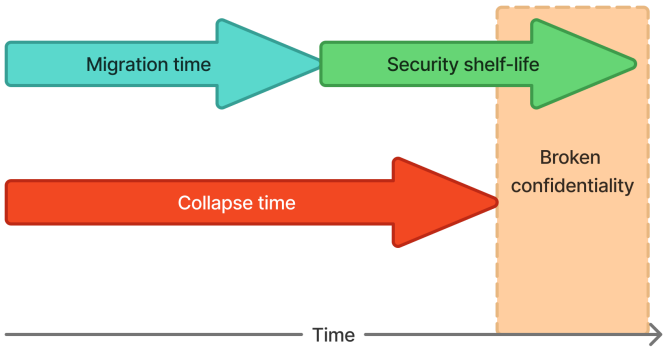

Figure 3.1: Mosca's theorem

As visualized in Figure 3.1, our confidentiality is broken if the sum of security shelf-life and migration time exceeds that of the collapse time [28]. In that scenario, the data we protect today will lose their privacy, while they are still valued.

Although quantum computers are still years from becoming cryptographically relevant, adversaries are already preparing.

In anticipation of future possession of a CRQC, an adversary can prepare by collecting transmitted data encrypted using classical schemes. This is a strategy known as *Harvest-Now-Decrypt-Later*.

If they eventually become in possession of this computing power, the privacy of the stored data is compromised. As revealed in the Snowden leaks, governments are already doing this.

This highlights the urgency in transitioning to post-quantum KEM schemes to preserve confidentiality past the arrival of CRQCs. Otherwise, the contents of the communication in which the keys used to encrypt could be revealed. Meanwhile, digital signatures only need to be replaced by before CRQCs arrive, as signatures quickly expire, and forging past signatures

is purposeless.

3.3.1 Current State in the Transition

Much of the trust in the classic cryptography we use today stems from its persistent security over the years. They have been continuous subjects of research and scrutiny and yet remain secure. Post-quantum schemes, on the other hand, are still relatively new and their robustness has therefore not yet been subjected to the same degree of cryptanalysis.

As we cannot yet fully trust that new attacks or vulnerabilities are discovered, many of them are deployed in *hybrid* designs. These combine post-quantum algorithms with classical ones, thus obtaining both the assurances of classical schemes and the post-quantum security of the new algorithms. Although hybrid approaches have clear security benefits, they also require the computational resources of both schemes.

Several big companies are already deploying PQC in their applications. Signal has implemented their solution, Post-Quantum Extended Diffie-Hellman (PQXDH), which aims to achieve quantum-resistance in a hybrid approach consisting of CRYSTALS-Kyber and DHKE [12].

Apple has announced PQ3 for their iMessage app, another hybrid design that combines Elliptic Curves with CRYSTALS-Kyber. However, they have reduced the frequency of key updates to preserve user experience [1].

Google is also transitioning to hybrid solutions. They offer ML-KEM security in the Chrome Browser and ML-DSA for Google Cloud Key Management Service [18]. They have, however, held back on this service for Android devices due to a noticeable increase in latency [56].

The transition to PQC has begun and the first standards are ready for implementation and use. The industry has already begun the process of securing their services and resources using PQC algorithms. Google, Apple, and Signal, presumably among many others, have already begun making substantial progress, paving the way for others to follow.

However, achieving a full transition remains a significant task. PQC must be implemented by many, and backward compatibility and interoperability with established internet protocols [24], crucial systems, as well as government and military infrastructure, may hinder seamless integration.

To aid in the process, several agencies seek to provide guidance. National government agencies are working with their respective industries for a

national transition. The US National Security Agency provides resources and guidance in hopes of achieving a full transition before 2035 [43]. In Norway, the Nasjonal Sikkerhetsmyndighet (NSM) operates under the assumption that systems should be prepared in the early 2030s and provide businesses with advice on how to migrate to post-quantum solutions [30].

Chapter 4

Messaging Layer Security

Messaging Layer Security is a protocol for tree based continuous group key exchange. MLS provides robust and trusted security while performing efficiently in groups where its participants can change over time.

Whereas the naïve approach visualized in Figure 2.10 scales linearly with the number of participants, the MLS protocol scales with the logarithm of the size of the group. Taking advantage of ratchet trees where members and keys are distributed in nodes, each encryption and key update only needs to be performed for certain paths in the tree. This effectively reduces the number of operations required from $N-1$ to $\lceil \log_2(N) \rceil$, where N is the size of the group. For larger groups, this results in substantial computational savings.

Adhering to the current standards in messaging such as the Signal protocol, MLS provides *asynchronous group key exchange*, forward secrecy, and post-compromise security. Asynchronous group key exchange enables key updates to occur in the absence of one or more group members. When the members return, they will perform their corresponding actions and synchronize with the updated group state to remain an active member of the group.

MLS itself is not a messaging application, but a security layer with emphasis on interoperability and extensibility. It is a secure and efficient protocol in which programmers can implement a high level of E2EE in their applications. The application design in its entirety must be careful not to introduce vulnerabilities outside of MLS. RFC 9750 [3] on the architecture of MLS provides additional guidance for those incorporating MLS in a

group messaging system for this reason.

MLS does not provide methods for initial authentication or message delivery. Any applications integrating MLS must resort to alternative solutions for this. MLS is designed to protect confidentiality and integrity in insecure delivery services (DS), but it depends on a trusted authentication service (AS) to provide the initial authentication of entities.

4.1 MLS Working Group

The MLS protocol was made an Internet standard in RFC9420 [2] in 2023. It is built and maintained by the MLS working group (MLSWG) of the Internet Engineering Task Force (IETF). The MLSWG continues to further develop the protocol and is planning on support for post-quantum adaptability by December 2026. Their approach is a hybrid design that aims to maintain efficiency while providing post-quantum security [57].

Several open-source implementations have already been developed in various programming languages, some of which have been tested and assessed by the MLSWG and can be found in their implementation list [27]. For this thesis, any practical experiments will be conducted in OpenMLS [51], an open-source library written in the Rust programming language.

4.2 Terminology and constructs specific to MLS

This thesis will follow the terminology used in RFC 9420 [2].

Entities in the MLS protocol are called *clients*. A *group* is a collection of clients with the secret keys required to communicate with each other. Clients with access to these keys are then *members* of the group. The lifetime of a group is divided into consecutive epochs. A group *epoch* describes the state of a group at that point in time and is unique for its membership and established keys. The members of a group in a given epoch share an *epoch secret*, which is only known by members and unique to that epoch. MLS ensures the confidentiality of the epoch secret. Whenever a change occurs to the group, it advances to the next epoch.

MLS messages are transmitted as *public* or *private* message types. Public messages are signed by the sender and contain information about the epoch and thus can validate the authenticity of the sender and the epoch it was sent

during. Private messages are also encrypted, thus ensuring confidentiality for members in that epoch. MLS members communicate using *application* messages that are transmitted as private messages. *Handshake* messages can be of either the private or public type and are used to transmit information that leads to an epoch change. This information comes in *proposals* and *commits*, which will be covered later. *KeyPackage* objects contain the identity, public key, and capabilities of a client and are used in the process of initiating them in an existing group. This lets the group verify its identity and determine compatibility with the group configurations prior to the inclusion of the new member.

Analogously, the client to be included receives a *WelcomeMessage* which contains all the information he will need to build the state of the group, including its *ratchet tree*, *secret tree* , and *key schedule*. These are signed, such that the newcomer can verify that he received authentic group information.

Members in a group can queue updates to the group state with *proposal* messages. These are appended to a *proposal list* and are applied when a subsequent *commit* occurs. Commit messages inform other group members to update their ratchet tree and key schedule according to the validated proposals in the proposal list, thus advancing the group into a new epoch. Commit messages are regularly triggered by certain conditions. For example, a commit must be submitted prior to transmitting any application messages if there are valid proposals in the proposal list. This ensures that any pending removal of members occurs before they gain access to the application message. Since commit messages provide important security benefits, members can perform *empty commits* to achieve these even when there are no pending proposals.

4.3 Essential Concepts

The MLS protocol is built on three essential concepts, the ratchet tree, the *secret tree*, and the *key schedule*.

The membership of groups are represented in a ratchet tree, in which members are assigned leaf nodes. Members reference this ratchet tree to authenticate other members and to encrypt messages to subsets of the group.

The key schedule is responsible for deriving keys and secrets used during the lifetime of an MLS group. For each epoch, the key schedule derives a

secret tree with the same structure as the group's ratchet tree from the epoch secret. Each leaf node in the secret tree corresponds to the member at the same node in the ratchet tree and contains a base secret used to initiate two sender ratchets. These are symmetric hash ratchets; a handshake ratchet to derive keys for handshake messages and an application ratchet to derive keys for application messages.

Each group member locally maintains partial instances of these three components that are required for them to function as a group member. Different parts of the MLS protocol are designed to ensure that legitimate members have complete and synchronized views of the components as the group proceeds in epochs.

There are three crucial actions performed by members to apply changes to the membership of a MLS group; *Add*, *Update*, and *Remove*. All these are initially sent as proposals and applied in commits.

To add a new member to the group, an existing member creates an Add proposal. The new member is then included in the next epoch of the group. To ensure forward secrecy, the cryptographic state of the group must provide the newcomer with fresh keys that cannot be used to decrypt past messages.

Similarly, a member can create an Update proposal to refresh the keys associated with him in the next epoch, thus providing forward secrecy for the keys in the path of the updated member.

Lastly, members can create a Remove proposal to evict another member from the group. The removed member will not receive the necessary secret material to derive the new epoch secret. This provides post-compromise security, as he will not be able to decrypt any messages related to the new epoch.

4.4 Protocol Flow in Creation of MLS Groups

The first member creates the MLS group in its initial epoch. To include additional clients, the creator must collect their published KeyPackages and use these to add them to the group. This process is visualized in Figure 4.1, where Alice retrieves the KeyPackage objects representing Bob and Charlie. This process is done by members creating Add proposals for new members in any state of existing groups.

Proceeding, Alice creates Add proposals to add Bob and Charlie to the

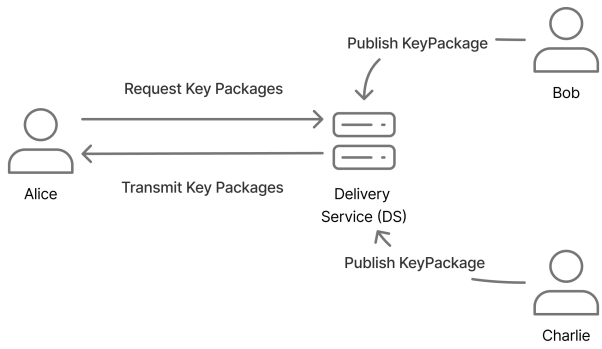

Figure 4.1: Alice retrieves Key Packages published by Bob and Charlie

group, which are appended to the current proposal list. Bob and Charlie will be added to the group upon the next commit. The commit will execute all valid pending proposals, which will be broadcast to any existing members as shown in Figure 4.2, instructing them to synchronize with the updated group state.

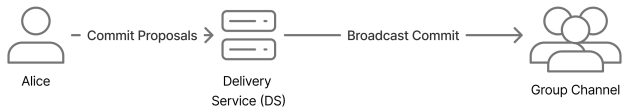

Figure 4.2: Alice commits pending Proposals

The member that commits the proposal list will update their view of the ratchet tree to include the new members, and generate and transmit WelcomeMessages for each of them. When they receive their WelcomeMessage, they use them to build the ratchet tree and key schedule of the group they are joining. At this point, the group has entered a new epoch in which pre-existing and newly added members all maintain the updated and synchronized cryptographic state of the MLS group.

4.5 Ratchet Tree Concepts

Ratchet trees are the most defining characteristic of the MLS protocol. The group members and keys are organized in these perfect binary tree data structures. The size of the ratchet tree will be adjusted during the lifetime of the group, by *extending* when there are no more vacant leaves to fit new members and *truncating* when a removal allows the tree to be halved in size. Extending and truncating ratchet trees always affect the number of leaves by a factor of 2.

The leaves in a ratchet tree represent members of the group, but these can also be *blank* when they do not contain information about a member. The intermediary nodes contain asymmetric keys in which the public ones are known by all members, but the private ones are only distributed according to the *tree invariant*.

Encryption and key updates can therefore efficiently achieve FS and PCS by performing actions for subtrees of the ratchet tree. A given member can encrypt to all but one of its co-members in $log_2(N)$ operations by encrypting to subtrees, which is what is done for the remove operation that provides post-compromise security. The same applies when a member advances his cryptographic contribution in the tree by deriving fresh keys to achieve forward secrecy. The new keys can be efficiently distributed by encrypting them to subtrees of the ratchet tree in $log_2(N)$ operations. Note that in MLS, the group size corresponds to the number of leaves in the ratchet tree.

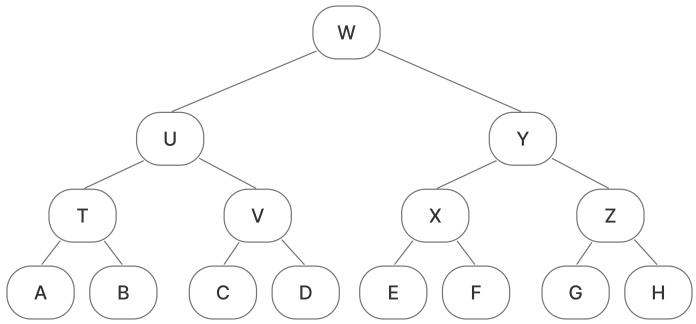

Figure 4.3: Ratchet tree for a group with eight members

For example, Figure 4.3 visualizes the encryptions performed by node

A in the removal of node F. This is be done by encrypting using the public keys in nodes U, E and Z, thus avoiding encrypting to any ancestor nodes of F. Note that PCS is not provided until the Remove proposal is performed in a commit. At this point, any existing members can continue the group's lifetime with a new epoch secret, which the removed member does not have access to.

4.5.1 Leaf Nodes

Non-blank leaves contain the public information regarding the group member at that leaf. It can be used to lookup supported cipher suites, protocol versions, extensions, and capabilities of that member, as well as its credentials and public keys. Only the member represented at the node knows its private keys. Other group members can use the public signature key at the node to verify any digital signatures produced by the member, and its credential authenticates the identity and signature key of the member. The member provides a signature of its leaf node contents, thus verifying its authenticity and integrity to co-members of the group. Valid signatures are unique to each epoch.

Leaf nodes are verified by the group when they appear in proposals and commits. In doing so, they ensure the uniqueness of its keys and compatibility with the group state. New members verify the leaf nodes of the group they are joining to ensure that the ratchet tree is the authentic tree of the group.

4.5.2 Parent Nodes

The *parent nodes* contain a public key in which the corresponding private key is exclusively known to the group members among its descendants. In particular, this is how MLS achieves its $\log_2(N)$ efficiency for encrypting messages, updating keys, and removing members. Members can encrypt to intermediary nodes, in which several members know the private key used to decrypt. The parent nodes also contain a *parent hash* and a list of *unmerged* leaves.

4.5.3 Blank Nodes

Blank leaf nodes are leaf nodes that do not represent a member. The number of empty leaves in a group corresponds to the number of new members that can be added without extending the ratchet tree of the group. Blank intermediary nodes are parent nodes that do not contain a key pair, which can occur after update and remove proposals are committed. Intermediate nodes are always blank if all their descendant nodes are blank.

4.5.4 The Tree Invariant

The tree invariant specifies the permitted distribution of private keys to group members. All group members have access to the ratchet tree and thus the public keys of all nodes in the tree. However, they are restricted to only knowing the private keys of their ancestor nodes. The *tree invariant* and is ensured by the protocol. Scenarios can occur where leaf nodes do not know the private key of an ancestor node, in which case they are *unmerged* in relation to that node.

4.5.5 Tree Hash and Parent Hash

All nodes in the ratchet tree contain both a *tree hash* and a *parent hash*. The tree hash is computed recursively from the leaves to the root and thus contains information about the subtree of the node. For leaf nodes, it is simply the hash digest of some information about the node. For intermediary nodes, it is the digest of some information about the node itself and the tree hash of both its children. This is visualized in Figure 4.4.

This computation continues recursively throughout the tree, resulting in the tree hash at the root being affected by the entire tree. Tree hashes are used to ensure that the group members agree on the current structure of the ratchet tree. New members use this to verify that the tree hash of the root of their ratchet tree matches the tree hash included in their received WelcomeMessage.

In contrast to tree hashes, parent hashes are computed from the root to the leaves. This attribute, in conjunction with the procedure for updating the parent hashes, makes them function as a proof that the nodes in the tree were set by legitimate members of the MLS group.

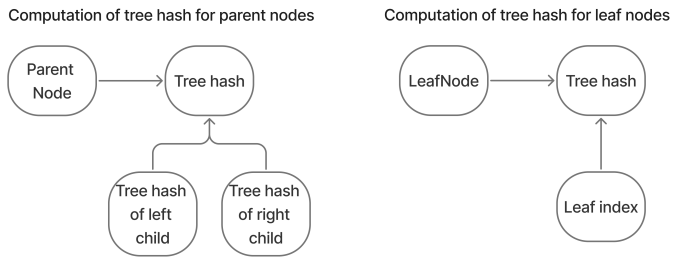

Figure 4.4: Computation of tree hashes

Each node stores the parent hash computed from its parent. The root does not store a parent hash as it has no parent to calculate it from, but parent hashes are computed at the root and stored in the children of the root. Note that the parent hash of a node is not affected by the node that stores it. The parent hash stored in a given node N is the parent hash of its parent P. It is computed from the parent hash stored at P, the public key of P, and the tree hash of P's child node S on the *copath* of N, as shown in Figure 4.5. Whenever a group member performs an action that updates keys, it must also recompute and set the parent hashes for the nodes in its direct path.

The parent hashes at the leaf nodes are signed by the member at the node, thus attesting to which keys they introduced to the tree and to whom they encrypted the keys. In this way, the keys of all intermediary nodes in the three can be traced back to the member who last updated them. By doing

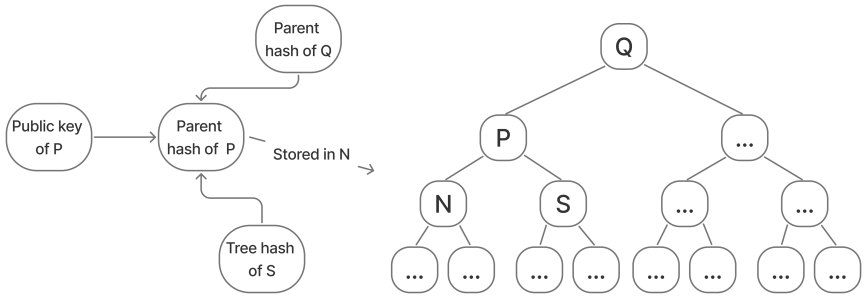

Figure 4.5: Computation of parent hashes

this and verifying the signature over the parent hash stored in the leaf, one verifies the authenticity of the keys. When nodes are verified in this way, they are considered *parent-hash valid*. Clients are required to check that all nodes are parent-hash valid when first joining a group, and existing members do it when processing commits.

4.5.6 Unmerged leaves

A leaf node is considered an *unmerged leaf* in relation to an ancestor if it does not know the private key of its ancestor. All parent nodes contain a list of unmerged leaves among their descendants. This list is used to ensure that any member intending to encrypt to the unmerged leaf is informed that the recipient does not know the private key at this depth of the tree. Proceeding, the sender knows that it must delve deeper to successfully encrypt to the unmerged member. Members are unmerged in relation to all ancestors when they first join a group and remain unmerged until it updates the keys of its ancestor nodes or until another node encrypts the private keys of these nodes to them.

4.5.7 Resolution of a Node

The *resolution* of a node points to the set of keys needed to encrypt to all of its non-blank descendants. For all blank nodes, it is the concatenation of the resolutions of its children, and for all others, it is a concatenation of the public key of the node and the public keys of any unmerged leaves at the node. As a result, the resolution of the root node consists of the public keys required to encrypt to all members of the group.

4.5.8 Paths in a Ratchet Tree

In addition to *direct paths*, which list the parent nodes required to traverse from a given node to the root, MLS additionally uses *copaths* and *filtered direct paths* in the protocol. The copath of a node lists the sibling of the node, as well as all siblings of nodes in its direct path. The filtered direct path of a node B, is the direct path of B that excludes all nodes that have a child on the copath of B with an empty resolution.

The filtered direct path is useful for members when updating keys. To update one's contribution to the ratchet tree and achieve forward secrecy and post-compromise security, the member must delete the keys in its direct path and update the keys in its direct filtered direct path.

For each updated node, their other descendant members must derive the new private key, and all members must update the public key of the node. The filtered direct path of the member that performs the update holds the information about the encryptions that need to be done to accomplish this distribution.

Recall that the resolution of a node points to the keys required to encrypt to all members in its subtree. If the resolution is empty for a node on the copath of the updating member, then setting the keys of its parent would result in keys that encrypt to the same set of members as an already updated node closer to the leaf of the member. The filtered direct path thus ensures that these kinds of duplicate key pairs are skipped when updating keys.

4.5.9 UpdatePath and Path Secrets

UpdatePath objects are used in commits to distribute updated keys to group members. They contain the information required for members to update the public and private keys in the ratchet tree according to the tree invariant. The public key is distributed to all members, while the *path secret* used to derive the private keys of a node is encrypted to the descendants of the node.

The member performing the update derives the key pair for its leaf using a private *leaf secret*. This leaf secret is additionally used to derive the first path secret. The first path secret can be used to derive the key pair of the closest updated ancestor, P. According to the tree invariant, this private key should be exclusively known to descendants of P. It is therefore encrypted to these descendants as they appear in the resolution of P and then transmitted to them in an UpdatePath.

Then, the updating member derives the next path secret from the first one. Like its predecessor, it is used to derive the key pair of the next ancestor node on the filtered direct path, Q. It is encrypted to the resolution of this node and is included in an UpdatePath. The descendants of P are also descendants of Q, but they are not included in its resolution. Instead, they derive this next path secret themselves and then use it to derive the private key of Q.

As shown in Figure 4.6, path secrets are produced for all updated key

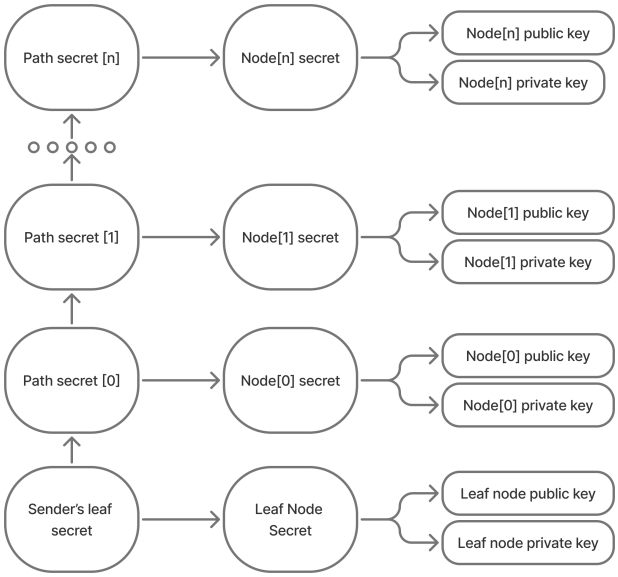

Figure 4.6: Derivation of path secrets and key pairs

pairs in the tree. Members receive path secrets for the first node they are to know the private key of and derive the remaining private keys themselves. This results in the distribution of private keys in a manner that adheres to the tree invariant while providing all group members with a full and synchronized view of the tree.

4.6 Ratchet Tree Operations

Let us now take a look at how some of these ratchet tree concepts are in effect in proposals and commits.

Proposal messages are made by members and effectively queue changes to the group state. The main proposal types covered in this section are add, update, and remove. All proposals are appended to the group's proposal list for the current epoch and are validated and applied by the next commit. Commit messages progress the cryptographic state of the group into a succeeding epoch, thus seeding an updated key schedule that provides a new secret tree.

Add proposals are made to introduce a new member to the group. When the Add is committed, a WelcomeMessage is created and transmitted to the new member, and it is assigned to the leftmost blank leaf node in the ratchet tree. For all non-blank nodes in the direct path of the member, the node will be appended to their list of unmerged leaves. In other words, the new member is not granted any private keys for the epoch in which it is first introduced to the group. It only gains access to these keys by performing a commit to set them, or when other members encrypt path secrets to them in subsequent commits.

When Update proposals are committed, the member that made the proposal blanks all the key pairs in its leaf node and direct path, thus providing the proposer with both post-compromise security and forward secrecy. The member derives a new key pair and includes the public key in the proposal to be distributed with the commit. However, only the key pairs of intermediate nodes in the filtered direct path of the committer are repopulated in the next epoch. By members regularly making Update proposals, the consequence of a compromised key is narrowed. Update proposals allow several members to achieve PCS and FS in a single commit performed by someone else.

Remove proposals are made to evict a member from the group and

achieve PCS by progressing the group to a cryptographic state where the evicted member does not know any of its updated secrets or private keys. This is done by blanking the leaf node and direct path of the removed client from the ratchet tree. New key pairs are produced for nodes in the filtered direct path of the member that commits the remove proposal.

There are three types of commit messages. The default configuration, *full commit* or just commit, applies the proposal list and updates the leaf node and the filtered direct path of the committer. Executing a commit provides forward secrecy by removing any pending evictions, and post-compromise security by updating the committers contribution to the group.

Members can still achieve these properties even when there are no pending proposals by sending the second type of commit, *empty commits*. *Empty commits* update the leaf and the filtered direct path of the committer without referencing any proposals.

The third type of commits is *partial commit*, which refers to a restricted set of proposal types but does not update any group keys. Since no keys are updated, partial commits cannot reference remove or update proposals, as that would blank essential keys and result in a group unable to communicate.

Commits are regularly triggered in the MLS protocol. For example, a member must commit before sending any application data. This ensures that any members referenced in remove proposals are expelled before gaining access to the transmitted data.

Forward secrecy and post-compromise security in the MLS protocol are provided to keys used by group members in committing Remove and Update proposals, as well as in full and empty commits.

Any member that refrains from updating its keys by performing one of these methods will be vulnerable until they do, as this expands the window for an attacker to compromise and benefit from the key during the time it remains active.

For this reason, long-term absent members should be removed from the group to mitigate the threat [2]. MLS leaves it up to the application to decide on rules for the frequency of key updates and removal of members who do not meet the requirements. A possible solution towards ensuring keys of all members are regularly updated is to implement a forced refresh frequency similar to the procedure in the Signal Protocol, where keys are updated every time a message is sent.

Chapter 5

Post-Quantum Secure Implementation of Messaging Layer Security

In this section, we start off with a walkthrough of the official draft by the MLS working group to efficiently provide post-quantum security in the MLS protocol. Their solution is a hybrid approach called the Amortized Post-Quantum (APQ) MLS Combiner. Then, we embark on the experiment for this thesis. An investigation into the performance impact of transitioning to post-quantum confidentiality in MLS by naively replacing classical methods for key encapsulation with post-quantum ones.

Lastly, we reflect on and measure our findings against recently published results by Britta Hale and Noah Greene in a paper that implements the APQ MLS combiner.

5.1 The Amortized Post-Quantum MLS Combiner

The MLS Working Group (MLSWG) is currently developing the Amortized Post-Quantum (APQ) Combiner [57]. According to their planned milestones for MLS, it is expected to be ready by December 2026. Our walkthrough is of its most recent draft.

The APQ-combiner is a hybrid design that achieves post-quantum con-

fidentiality and authenticity while amortizing the computational costs of its PQC algorithms by combining a post-quantum (PQ) MLS session with a classical MLS session.

Two MLS groups are created, one that uses a classical cipher suite as defined in RFC 9420, and one that uses a PQ MLS cipher suite as defined in *MLS Cipher Suites with ML-KEM* [23]. These two sessions are run in parallel with synchronized group memberships. By regularly injecting secrets from the PQ session into the classical session, it inherits post-quantum guarantees.

This is done using an additional proposal type, *PreSharedKey*. The PreSharedKey proposal allows for a pre-shared key (PSK) from an external source to be incorporated in the key schedule and thus affect the group secrets it produces. As in hybrid designs in general, the classical session provides well-trusted assurances from its persistent security over the year.

The combiner is flexible in the way that the communication system that implements it can choose the frequency in which the computationally demanding PQ key updates should occur.

The APQ MLS differentiates between partial and full commits to amortize the frequency of operations in the PQ session. To ensure that both group sessions maintain synchronized memberships, the combiner requires full commits when performing Add and Remove proposals. Full commits advance the epoch of both group sessions and trigger a PreSharedKey proposal to inject a fresh PQ-PSK into the classical key schedule.

Doing so provides PCS and FS to the classical key schedule against both quantum and classical adversaries. Members of the group can then communicate using the cryptographic keys produced from the classical session, which has post-quantum secure properties due to its injected PQ-PSK.

All other proposals can be performed using partial commits, which will only advance the epoch of the classical MLS session. As group membership are unchanged, no synchronization is necessary. The key schedule of the classical session will continue to produce keys with post-quantum confidentiality and post-quantum forward secrecy even as partial commits are executed. However, the PQ post-compromise security of keys is achieved only when a new full commit is performed.

In the draft, two modes of operation are specified. The **Confidentiality only mode** replaced classical methods for key encapsulation with the new post-quantum standard ML-KEM. This protects against Harvest Now, De-

crypt Later attacks, but will be vulnerable to forgery attacks when CRQCs eventually arrive. The **Confidentiality and authenticity mode** protects against both, by implementing the ML-KEM and PQC digital signatures using ML-DSA.

5.2 OpenMLS and the XWING cipher suite

The OpenMLS [51] library is an open-source implementation of the MLS protocol developed and maintained in the *Rust* programming language by Phoenix R&D and Cryspen. OpenMLS has a comprehensive user manual and documentation[50] and is a listed implementation by the MLSWG[27]. In combination, that provided reasonable assurances to proceed in using it for this thesis. This experiment uses OpenMLS version 0.7.1 [49].

In version 0.6, OpenMLS added support for the use of some post quantum cipher suites. These use *XWING*, a hybrid post-quantum key encapsulation method. XWING consists of ML-KEM-768 and X25519, an elliptic curve DHKE KEM [9]. The ML-KEM implementation used is developed and formally verified by Cryspen [5, 4, 21]. XWING provides post-quantum confidentality and thus protection against Harvest Now, Decrypt Later attacks.

OpenMLS developers report that larger keys and messages lead to an increased workload when using XWING [21].

5.3 Comparison of Post-Quantum and Classical MLS Sessions

For the upcoming experiment, the two ciphersuites:
`MLS128_DHKEMX25519_CHACHA20POLY1305_SHA256_Ed25519` and
`MLS256_XWING_CHACHA20POLY1305_SHA256_Ed25519` are used. They should only differ in the cryptographic algorithm used for KEM.

Measurements have been made using the Criterion library [15]. A separate script for each cipher suite was developed for the measurements recorded in this thesis. The scripts closely follow the OpenMLS user manual[50] when possible.

The two cipher suites are provided by separate cryptographic crates,

RustCrypto[48] and LibCrux[47], respectively. Their implementations may vary slightly. An effort has been made to minimize the differences between the two scripts.

All scripts for this thesis were executed on a 64bit Ubuntu(v24.04) Virtual Machine, allocated 23894 MB memory and 6 processors. The host device has an AMD Ryzen 5 7600 6-Core 12-Thread 3801Mhz CPU, and 32GB DDR5 6000Mhz RAM.

The scripts are functional, but can likely be optimized further. However, their purpose in this thesis is to investigate the costs in memory and speed by upgrading to PQC, for which estimates are acceptable.

5.3.1 Setup

For both cipher suites, each operation has been measured for group sizes of 10, 50, 100, 200, 500, and 1000. We intentionally exclude measuring the creation of groups. While commit messages are frequent in continuous groups, their creation is a one-time cost. Therefore, the group members are generated and added to an MLS group before any measurements start.

Six different operations are being measured in this simulation: **add member**, **remove member**, **update member**, **join group**, **create commit**, and **process commit**. Before measuring them, an MLS group is created with N members.

For **add member**, **remove member** , and **self update** we measure the time it takes the committer to atomically propose and commit the operation, as well as for all remaining group members to synchronize to the contents in the commit. In other words, the time it takes all group members to sequentially advance from one epoch to another. However, in practice, the computational load is distributed among the group members and processed in parallel. These measurements are recorded for a freshly created group where all members except the creator are unmerged in relation to their ancestor nodes. As they do not have access to private keys, the committer must encrypt path secrets to all members individually. In continuous groups, the committer encrypts to the resolutions of the $\lceil \log_2(N) \rceil$ ancestor nodes on its filtered direct path.

The **Self update** is an empty commit. Whereas the update proposals in the MLS protocol are proposed by one member and committed by another, a self update in OpenMLS blanks and updates the keys in the path of the

committer. This operation is analogous to an empty commit in the MLS protocol.

The **Add member** performs the commit which must additionally create a welcome message for the new member. We do not measure the client joining the group here.

The **Remove member** performs the commit and must also blank the removed members' leaf and path in the ratchet tree.

Create commit, **process commit** , and **join group** are more fine-grained measurements aiming to capture the computations performed by a single group member. These highlight individual actions performed when participating in an MLS group.

Create commit measures a member creating an empty commit and advancing into an epoch where the contents of the commit are applied. As it references no proposals, the commit simply updates the keys in the committers path.

Process commit measures a group member processing a received commit and advancing epochs.

Join group measures a client processing a received Welcome message to become a group member. Note that joining a group entails building the ratchet tree and key schedule of the group, but as they are unmerged in relation to all ancestors, they do not process any private keys.

The benchmarking scripts each sample the measured operations in 50 iterations. Each operation shows noticeable variations in the samples. Figures 5.1 and 5.2 were created by the Criterion tool when running the scripts. They show the distribution of execution times for the 50 samples of **Add member** to a PQC session of 1000 members and to a classical session of 50 members, respectively. Performance metrics for each operation are recorded in the average time of its samples.

We also measure the byte sizes of commit messages, ratchet trees, welcome messages, and key packages that are transmitted during benchmarked MLS operations.

The recorded sizes of the messages are from a single sample for each group size and cipher suite. The ratchet trees are sampled after group setup. The welcome message is sampled when it is unpacked in its recipient. The commit messages are sampled when they are first created by the committer. Key packages are sampled when created.

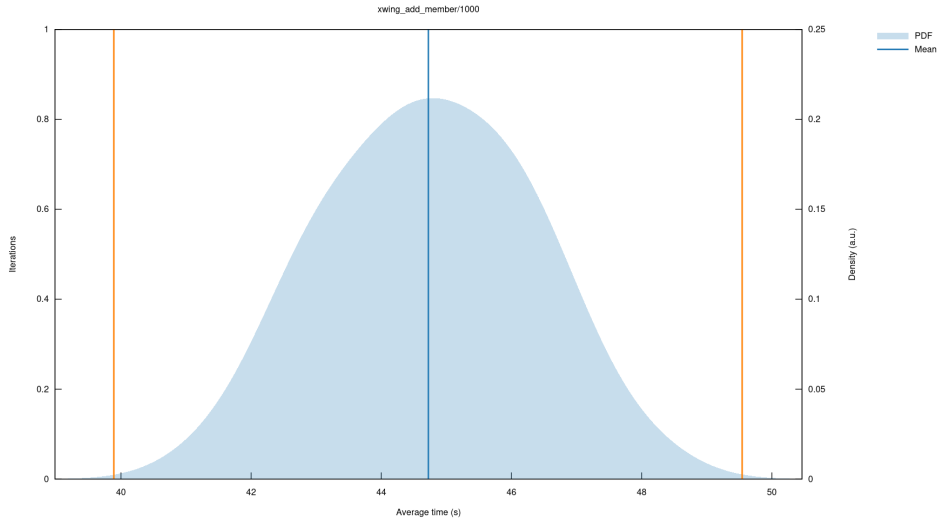

Figure 5.1: Density in PQ MLS session: Add Member in a group of 1000 members

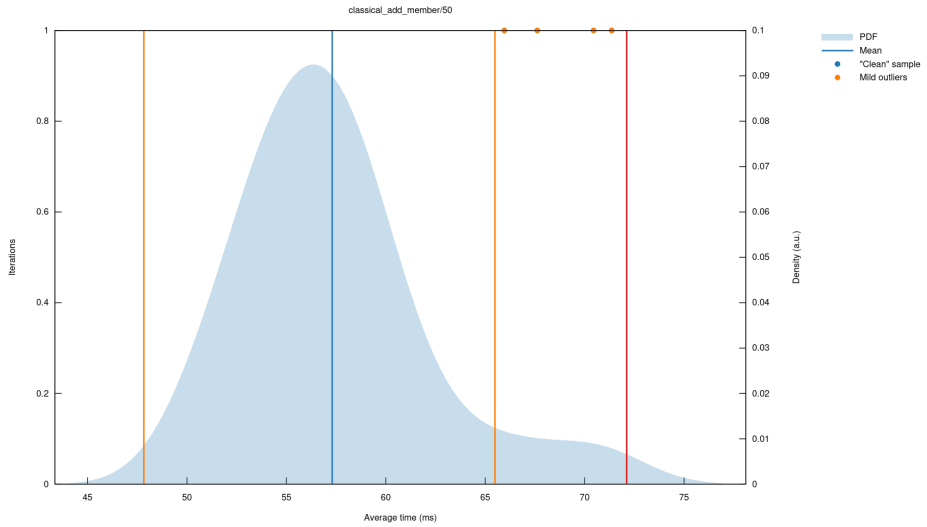

Figure 5.2: Density in classical MLS session: Add Member in a group of 50 members

5.3.2 Packet Sizes

A considerable portion of the overhead in PQ algorithms stems from their increased key and ciphertext sizes. The following presents the recorded byte sizes of key packages, welcome messages, ratchet trees, and commits for each of the measured MLS operations in both classical and post-quantum MLS sessions.

The key packages contain information about the client to which they belong and are not affected by the size of the group. It is fixed in size for each cipher suite. In our chosen cipher suites, classical key packages are **286 bytes** and PQ key packages are **2656 bytes**. This corresponds to an increase of approximately $2656/286 \approx$ **9.3** times when upgrading to PQ public keys.

Table 5.1: Classical Message Sizes (bytes)

Group Size	Ratchet Tree	Add Commit	Welcome	Self Update	Remove Commit
10	1851	1544	2668	1258	1183
50	9291	4894	10244	4608	4533
100	18593	9029	19620	8743	8668
200	37293	17268	38388	16982	16907
500	93393	41905	94556	41619	41544
1000	186893	82942	188124	82656	82581

Table 5.2: Post-Quantum Message Sizes (bytes)

Group Size	Ratchet Tree	Add Commit	Welcome	Self Update	Remove Commit
10	13701	19642	21540	16986	15820
50	68543	68928	78886	66272	65108
100	137093	128700	148689	126044	124880
200	274293	247022	287142	244366	243202
500	685893	599544	699995	596888	595724
1000	1371893	1186266	1387248	1183610	1182446

Within each cipher suite, there are negligible differences between the sizes of the three commit messages in equal sized groups. Looking at the values for the three commit operations in Tables 5.1 and 5.2 , we see that their differences roughly correspond to the size of a key package for all group sizes:

Add − Remove ≈ Key Package
Add − Update ≈ Key Package.

This corresponds to the MLS specification, where an Update is an empty commit, and an Add commit additionally refers to the new member by its Key Package.

As specified in RFC9420, the ratchet tree of a group always extends and truncates by a factor of 2. Each iteration in the size of the group between 50 and 1000 extends the ratchet trees once. The new tree structure contains double the number of leaves. When increasing from $N = 10$ to $N = 50$, the tree is extended twice, resulting in quadruple size.

Indications of this kind of increase can be seen in both tables when comparing messages by comparing group sizes. For example, the pattern is visible for all message types when comparing the group size 100 and 200 in Table 5.2. Meanwhile, in increasing from 200 to 500, the tree is extended once, but it also contains more than double the number of non-blank nodes, thus causing a somewhat higher increase from its additional public keys.

There is a high correlation between the sizes of the welcome messages and the ratchet trees. This is expected as the welcome message contains the information needed for a new member to initialize the key schedule and the ratchet tree of the group. The information needed to build the key schedule seems to grow much slower and become a lesser factor in larger groups. From Tables 5.1 and 5.2, it seems that while the ratchet tree grows, the information to build the key schedule is fixed at approximately 1000 and 10000 bytes for the classical and PQ sessions, respectively.

Table 5.3: Message Size Ratio ($Ratio$ = PQ Size/Classical Size)

Group Size	Ratchet Tree	Add Commit	Welcome	Self Update	Remove Commit
10	7.40	12.72	8.07	13.50	13.38
50	7.38	14.09	7.70	14.38	14.37
100	7.37	14.25	7.58	14.41	14.41
200	7.35	14.30	7.48	14.39	14.39
500	7.34	14.31	7.40	14.34	14.34
1000	7.34	14.30	7.37	14.32	14.32

Table 5.3 shows the relationship between the sizes in PQ versus classical sessions, calculated as **PQ size** / **Classical size**. This shows a clear pattern where the increase in message size when upgrading to PQ is mostly consist-

ent for all message types across group sizes. PQ commits are consistently approximately 14 times larger than classical commits, and the ratchet tree and welcome are consistently approximately 7 times larger.

This increase in size is significant, especially when considering number of encryptions for some messages, such as commits, scale with $log_2(N)$ of the group size, N. In 1000 member MLS groups, this requires encrypting to the resolutions of a maximum of $\lceil \log_2(1000) \rceil = 10$ ancestor nodes. Using the naïve approach described in Chapter 2, it would require encrypting to all 999 public keys of the group members.

5.3.3 Timing Benchmarks

The six measures are logically divided into two groups: **Add**, **remove**, and **update** measure the time complexity for all members to **collectively** advance epochs in a sequential manner. **Create** commit, **process** commit, and **join** group measure the time for a single member to **individually** enter a new epoch. These individual operations should correspond to the actual latency that a member in an MLS communication system will experience. The execution times for the operations in the various group sizes are recorded in seconds in Table 5.4 and Table 5.5, respectively.

Table 5.4: PQ Benchmarks (s)

Group Size	Add	Remove	Update	Commit	Process	Join
10	0.019	0.01937	0.01912	0.00373	0.0017	0.0023
50	0.15305	0.1906	0.16954	0.00836	0.00375	0.00835
100	0.46359	0.62226	0.54557	0.01554	0.01079	0.01849
200	1.7866	2.142	1.9329	0.0407	0.02517	0.04596
500	11.484	12.921	11.056	0.16472	0.13481	0.16583
1,000	44.723	51.086	48.182	0.57157	0.53247	0.5602

For both sessions, Tables 5.4 and 5.5 show similarities between the three different commits, and also a vast gap between the execution times of the individual and collective measurement.

The execution time of an Update for a given group size N should correspond to the sum of one member performing Create and $N-1$ members performing Process. From the values in 5.4 for a PQ session with 10 members,

Table 5.5: Classical Benchmarks (s)

Group Size	Add	Remove	Update	Commit	Process	Join
10	0.01169	0.00945	0.01013	0.00222	0.00081	0.00081
50	0.0573	0.05357	0.05512	0.00453	0.00202	0.00347
100	0.14271	0.13722	0.14106	0.00926	0.0047	0.00799
200	0.44709	0.41601	0.41272	0.02423	0.01558	0.02256
500	2.1563	1.9726	1.9968	0.11496	0.10828	0.11811
1,000	8.3903	7.9544	7.8794	0.43561	0.42363	0.44175

the Update takes 0.01912 seconds, while the individual actions evaluates to $0.00373 + 9(0.0017) = 0.01903$ seconds.

However, when looking at larger groups, for example, a PQ session with 1000 members, these do not correspond. The collective Update takes 48.182 seconds, while its individual actions evaluate to $0.57157 + 999(0.53247) = 532.5091$ seconds. This is because we measure the **worst case** scenario of processing a commit. In our script, the recorded member to process the commit is the sibling of the member that created the commit. Thus, the processing member receives the first path secret and must use it to compute all the private keys that are updated in the commit. The sibling of the committer is the only member that must compute all the updated private keys set by the committer, which in this exact case corresponds to all 10 private keys of its ancestors. In this Update scenario, there are 488 nodes that only compute the private key of the root node. 2^1 members compute 9 keys, 2^2 members compute 8 keys, 2^3 members compute 7 keys, and so on. This should be the explanation for the discrepancy between the measurements of the Update and the evaluation of its individual sub-actions.

The measured commit types are expected to provide similar execution times, as they are only set apart by small additional computations, such as creating a welcome message in the Add commit. Their main computational burden comes from updating keys in their path and distributing them to other members. Tables 5.5 and 5.4 show that the three types are all relatively close in execution times for both cipher suites. However, it is somewhat strange that Add is the fastest commit in the PQ session. Update is the only empty commit, whereas the Add and Remove perform additional actions. I do not see a clear explanation for why this occurs. The Update is, as expected, the fastest commit in the classical session.

For individual measurements, it seems that creating a commit is more expensive than processing a commit in both sessions. However, the difference between them becomes insignificant as the groups grow larger. Even though the member joining the group does not set any private keys in the tree, they must process the received Welcome to set up their ratchet tree of members and public keys. Joining a group shows execution times similar to that of creating a commit.

Table 5.6: Relative Performance Ratio

Group Size	Add	Remove	Update	Process	Create	Join
10	1.63	2.05	1.89	2.10	1.68	2.84
50	2.67	3.56	3.08	1.86	1.84	2.41
100	3.25	4.53	3.87	2.30	1.68	2.31
200	4.00	5.15	4.68	1.62	1.68	2.04
500	5.33	6.55	5.54	1.25	1.43	1.40
1000	5.33	6.42	6.11	1.26	1.31	1.27

The performance ratios in Table 5.6 are calculated as follows:

$$R_{i,j} = \frac{T_{\text{PQ},i}(N_j)}{T_{\text{Classical},i}(N_j)} \quad \text{where } i \text{ refers to operation type} \tag{5.1}$$

These refer to the relative performance ratios when upgrading from classical to PQ. For example, in group size $N = 200$, an Add performs approximately 4.00 times slower in a PQ session than in a classical session.

When comparing PQ and classical execution times, there is a stark difference between individual and collective measurements. From Table 5.6, the individual measurements show a relatively small increase in cost in upgrading to PQ, while the collective measurements show a vast increase. In other words, the time it takes for the group to collectively advance to the next epoch is much slower in the PQ session, while individual operations are less affected.

For the two largest group sizes measured, the execution times of the PQ session for collective operations are consistently between 5 to 7 times that of the classical.

At the same time, Table 5.6 shows that the performance of the PQ session is much closer to that of the classical session for individual operations.

Overall, the individual measurements show a decreasing trend in the relative increased cost of a PQ session as the groups grow.

An Update takes 1.89 longer in a 10 member PQ session and 6.11 times longer in a 1000 member PQ session compared to a classical session. At the same time, creating and processing a commit in a PQ session takes approximately 2 times longer or less.

A possible explanation is that the larger PQ keys are efficiently established but costly to distribute in key updates. Google has previously reported that in their post-quantum solution for the Chrome browser they experience efficient key exchanges but slow transmission of keys [56].

Another plausible cause is inaccurate measurements. One of the OpenMLS developers has previously raised an issue about unexpected additional overhead when benchmarking with Criterion[13]. In addition, Hale and Greene claim that it was not possible to separate the group setup from sustained operations in their post-quantum MLS implementation [14]. They also used the OpenMLS library and Criterion for benchmarking.

5.4 Additional Comparison

Very recently, in mid October 2025, Britta Hale and Noah Greene published a paper where they implement and measure CPU cycles, bytes, and runtime in an approximation of MLS's Amortized PQ Combiner[14]. The paper describes a demanding task in integrating post-quantum security into the current state of OpenMLS and its components. The APQ Combiner implementation uses the PQ/T Confidentiality Only mode, meaning they have integrated ML-KEM but not ML-DSA.

Their methodology differs from ours. Whereas we setup a group and measure a single operation in that group, they measure the continuous operations of a group iterating 500 epochs. In addition, they measure smaller group sizes in the range from 2 to 100. Instead of using XWING for ML-KEM, they have implemented the predecessor of ML-KEMs, Kyber, themselves.

Their results show significant reductions in CPU cycles, transmitted data bytes, and average runtime when comparing a combined APQ session with a pure PQ session.

In APQ configurations that inject PQ-PSKs every 10 commits, their results show that the combiner provides PQ confidentiality close to 30% faster

than that of a pure PQ session. This configuration also reduces transmitted bytes with 79%. By only performing PQ-PSK injections every 100 commits, the speedup increases to 48% and the reduction in transmitted bytes is 89% compared to that of a pure PQ session.

Chapter 6

Conclusion

The expected advances in quantum computing necessitate a post-quantum transition in cryptography. The Harvest Now, Decrypt Later attack further emphasizes the need to complete this transition as soon as possible. Mosca's theorem highlights that we should already be securing the confidentiality of long-term secrets with post-quantum cryptography.

Fortunately, the post-quantum transition is well underway. The National Institute of Standards and Technology has already established the first standards to protect against cryptographically relevant quantum computers. Big enterprises such as Google, Apple, and Signal have already begun to transition their services to provide this defense.

For a long time, secure group messaging services have been computationally expensive. The Messaging Layer Security protocol provides a solution to this. Using ratchet trees to achieve efficient key exchange for continuous groups while maintaining strong security, the MLS protocol provides the scalability to support the post-quantum transition for group messaging.

This thesis shows that while libraries already exist to integrate post-quantum standards in Messaging Layer Security implementations today, they do entail larger messages and increased computational workloads for large groups. However, the Amortized Post-Quantum Messaging Layer Security Combiner has been shown to substantially reduce this cost.

Bibliography

[1] Apple Security Engineering and Architecture. iMessage with PQ3: The New State of the Art in Quantum-Secure Messaging at Scale. https://security.apple.com/blog/imessage-pq3/, 2024. Posted February 21, 2024; accessed: 06 Oct. 2025.

[2] R. Barnes, B. Beurdouche, R. Robert, J. Millican, E. Omara, and K. Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, https://www.rfc-editor.org/info/rfc9420, July 2023.

[3] B. Beurdouche, E. Rescorla, E. Omara, S. Inguva, and A. Duric. The Messaging Layer Security (MLS) Architecture. RFC 9750, https://www.rfc-editor.org/info/rfc9750, Apr. 2025.

[4] K. Bhargavan, L. Franceschino, F. Kiefer, and G. Tamvada. Verifying Libcrux's ML-KEM. https://cryspen.com/post/ml-kem-verification/, Jan 2024. Accessed: 2025-11-18.

[5] Bhargavan, Karthikeyan and Kiefer, Franziskus and Tamvada, Goutam. Verified ML-KEM (Kyber) in Rust. https://cryspen.com/post/ml-kem-implementation/, Jan 2024. Accessed: 2025-11-18.

[6] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. Report on Post-Quantum Cryptography. Nist internal report (nistir) 8105, National Institute of Standards and Technology (NIST), Gaithersburg, MD, 2016. Accessed: 6 Oct. 2025.

[7] K. Cherkaoui Dekkaki, I. Tasic, and M.-D. Cano. Exploring Post-Quantum Cryptography: Review and Directions for the Transition Process. *Technologies*, 12(12), 2024.

[8] T. Comer. Poland's Decisive Role in Cracking Enigma and Transforming the UK's SIGINT Operations. Commentary, Royal United Services Institute (RUSI), , https://www.rusi.org/explore-our-research/publications/commentary/polands-decisive-role-cracking-enigma-and-transforming-uks-sigint Jan 2021. Accessed: 6 Oct 2025.

[9] D. Connolly, P. Schwabe, and B. Westerbaan. X-Wing: general-purpose hybrid post-quantum KEM. https://datatracker.ietf.org/doc/draft-connolly-cfrg-xwing-kem/09/, Sept. 2025. Work in Progress.

[10] Crypto Museum. The Enigma Machine—History. https://www.cryptomuseum.com/crypto/enigma/hist.htm. Accessed: 6 Oct. 2025.

[11] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[12] Ehren Kret and Rolfe Schmidt. PQXDH — Signal Protocol Specification. https://signal.org/docs/specifications/pqxdh/, 2025. Accessed: 6 Oct. 2025.

[13] Franziskus Kiefer (issued by). "issue #475 – criterion.rs. GitHub issue, https://github.com/bheisler/criterion.rs/issues/475, May 2021. Accessed: 2025-11-18.

[14] N. Greene and B. Hale. Making Post Quantum Key Exchange Efficient: An Implementation with the MLS Protocol. Cryptology ePrint Archive, Paper 2025/1881, , https://eprint.iacr.org/2025/1881, 2025.

[15] B. Heisler. Criterion.rs: Statistics-driven benchmarking library for rust. https://github.com/bheisler/criterion.rs, 2025. Accessed: 2025-11-18.

[16] Y. IGARASHI. Secret Writing in Ancient Civilization. In *Computing: a historical and technical perspective*. CRC Press, 2017.

[17] J. P. Indrøy. Selected Topics in Cryptanalysis of Symmetric Ciphers. https://bora.uib.no/bora-xmlui/bitstream/

handle/11250/2825441/archive.pdf, 2021. Open access; accessed: 6 Oct. 2025.

[18] Jennifer Fernick and Andrew Foster. Announcing quantum-safe digital signatures in Cloud KMS. https://cloud.google.com/blog/products/identity-security/announcing-quantum-safe-digital-signatures-in-cloud-kms, 2025. Published 21. Feb. 2025; accessed: 6 Oct. 2025.

[19] N. Kaleyski. Hash Functions, 2023. Lecture slides for INF143A: Anvendt Kryptografi, Universitetet i Bergen.

[20] N. Kaleyski. Message Authentication Codes. Lecture slides, INF143A: Anvendt Kryptografi, Universitetet i Bergen, 2023.

[21] F. Kiefer. Post-quantum openmls. https://cryspen.com/post/pq-openmls/, Apr 2024. Accessed: 2025-11-18.

[22] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.

[23] R. Mahy and R. Barnes. ML-KEM and Hybrid Cipher Suites for Messaging Layer Security. https://datatracker.ietf.org/doc/draft-ietf-mls-pq-ciphersuites/01/, Nov. 2025. Work in Progress.

[24] G. S. Mamatha, N. Dimri, and R. Sinha. Post-Quantum Cryptography: Securing Digital Communication in the Quantum Era. arXiv preprint arXiv:2403.11741 https://arxiv.org/abs/2403.11741, 2024.

[25] Mavroeidis, Vasileios and Vishi, Kamer and Zych, Mateusz D. and Josang, Audun. The Impact of Quantum Computing on Present Cryptography. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 9(3), 2018. Accessed: 2025-11-16.

[26] D. McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116, , https://www.rfc-editor.org/info/rfc5116, Jan. 2008.

[27] Messaging Layer Security Working Group (MLSWG). Implementation List — MLS Implementations. https://github.com/mlswg/mls-implementations/blob/main/implementation_list.md, 2025. Accessed: 2025-11-17.

[28] M. Mosca. Cybersecurity in an era with quantum computers: will we be ready? Cryptology ePrint Archive, Paper 2015/1075, https://eprint.iacr.org/2015/1075, 2015.

[29] R. S. Moxie Marlinspike. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/, 2025. Accessed: 2025-11-16.

[30] Nasjonal sikkerhetsmyndighet (NSM). NSM Cryptographic Recommendations, Version 2.0. https://nsm.no/getfile.php/1314334-1742808614/NSM/Filer/Dokumenter/Veiledere/NSM%20Cryptographic%20Recommendations%202025.pdf, 2025. Approved 17 March 2025; accessed: 6 Oct. 2025.

[31] National Academies of Sciences, Engineering, and Medicine. *Quantum Computing: Progress and Prospects*. The National Academies Press, Washington, DC, 2019.

[32] National Bureau of Standards. Data Encryption Standard (FIPS 46-1). Federal Information Processing Standard Publication 46, National Bureau of Standards, 1977. Issued 15 Jan. 1977; superseded 22 Jan. 1988; accessed 19. Nov. 2025.

[33] National Institute of Standards and Technology. Advanced Encryption Standard (FIPS 197). Federal Information Processing Standard (FIPS) 197, National Institute of Standards and Technology, 2001. accessed: 6 Oct. 2025.

[34] National Institute of Standards and Technology. FIPS PUB 197: Advanced Encryption Standard (AES). Technical report, National Institute of Standards and Technology, Nov. 2001. Issued November 26, 2001; accessed: 6 Oct. 2025.

[35] National Institute of Standards and Technology. Module-Lattice-Based Digital Signature Standard (FIPS 204). Federal Information Processing

Standard (FIPS) 204, National Institute of Standards and Technology, 2024. Accessed: 6 Oct. 2025.

[36] National Institute of Standards and Technology. Module-Lattice-Based Key-Encapsulation Mechanism Standard (FIPS 203). Federal Information Processing Standard Publication (FIPS) 203, National Institute of Standards and Technology, 2024. Accessed: 6 Oct. 2025.

[37] National Institute of Standards and Technology. Stateless Hash-Based Digital Signature Standard (FIPS 205). Federal Information Processing Standard (FIPS) 205, National Institute of Standards and Technology, 2024. Accessed: 6 Oct. 2025.

[38] National Institute of Standards and Technology. NIST Selects HQC as Fifth Algorithm for Post-Quantum Encryption. https://www.nist.gov/news-events/news/2025/03/nist-selects-hqc-fifth-algorithm-post-quantum-encryption, 2025. Published 11 March 2025; accessed: 6 Oct. 2025.

[39] National Institute of Standards and Technology (NIST). NIST Policy on Hash Functions. https://csrc.nist.gov/projects/hash-functions/nist-policy-on-hash-functions, 2022. Created January 04, 2017, Updated September 09, 2024, Accessed November 19, 2025.

[40] National Institute of Standards and Technology (NIST). Cryptographic hash function — Glossary Term. https://csrc.nist.gov/glossary/term/cryptographic_hash_function, 2025. Accessed: 15. Nov. 2025.

[41] National Institute of Standards and Technology (NIST). Post-Quantum Cryptography Standardization. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization, 2025. Created January 03, 2017, Updated November 19, 2025, Accessed November 19, 2025.

[42] National Institute of Standards and Technology (NIST). Selected Algorithms — Post-Quantum Cryptography. https:

//csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms, 2025. Accessed: 2025-11-17.

[43] National Security Agency. CSI – Commercial National Security Algorithm Suite 2.0 and Quantum Computing FAQ. https://media.defense.gov/2022/Sep/07/2003071836/-1/-1/0/CSI_CNSA_2.0_FAQ_.PDF, Dec. 2024. Accessed: 2025-11-17.

[44] R. Niederhagen and M. Waidner. Practical post-quantum cryptography. https://www.sit.fraunhofer.de/fileadmin/dokumente/studien_und_technical_reports/Practical.PostQuantum.Cryptography_WP_FraunhoferSIT.pdf, 2017. Accessed: 6 Oct. 2025.

[45] NIST Computer Security Resource Center. Post-Quantum Cryptography. https://csrc.nist.gov/projects/post-quantum-cryptography, 2025. Accessed: 19 Nov. 2025.

[46] NIST Computer Security Resource Center. Security (Evaluation Criteria) — Post-Quantum Cryptography Standardization. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-(evaluation-criteria), 2025. Accessed: 6 Oct. 2025.

[47] OpenMLS Authors. openmls_libcrux_crypto: A crypto backend for OpenMLS based on libcrux. https://crates.io/crates/openmls_libcrux_crypto, 2025. Accessed: 2025-11-18.

[48] OpenMLS Authors. openmls_rust_crypto: Crypto backend for OpenMLS using RustCrypto primitives. https://crates.io/crates/openmls_rust_crypto, 2025. Accessed: 2025-11-18.

[49] OpenMLS Contributors. Openmls: A rust implementation of the messaging layer security (mls) protocol. https://github.com/openmls/openmls, 2025. Accessed: 2025-11-18.

[50] OpenMLS Developers. Openmls book. https://latest.openmls.tech/book/, 2025. Accessed: 2025-11-18.

[51] OpenMLS Project. OpenMLS: An Open Source Implementation of the Messaging Layer Security (MLS) Protocol. https://openmls.tech/, 2025. Accessed: 2025-11-17.

[52] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Nature, Berlin, Heidelberg, 1 edition, 2009.

[53] R. Perlner. FIPS 206 Status Update. https://csrc.nist.gov/csrc/media/presentations/2025/fips-206-fn-dsa-(falcon)/images-media/fips_206-perlner_2.1.pdf, 2025. Accessed: 2025-11-17.

[54] J. H. Pierre-Alain Fouque et al. Falcon. https://csrc.nist.gov/CSRC/media/Presentations/Falcon/images-media/Falcon-April2018.pdf, 2018. Presented at First PQC Standardization Conference, 12 April 2018; accessed: 6 Oct. 2025.

[55] P. W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997. (revised v2, Jan. 1996).

[56] The Chromium Blog. Advancing Our Amazing Bet on Asymmetric Cryptography. https://blog.chromium.org/2024/05/advancing-our-amazing-bet-on-asymmetric.html, 2024. Published 23 May 2024; accessed: 6 Oct. 2025.

[57] X. Tian, B. Hale, M. Mularczyk, and Joël. Amortized PQ MLS Combiner. Internet-Draft draft-ietf-mls-combiner-02, Internet Engineering Task Force, Oct. 2025. https://datatracker.ietf.org/doc/draft-ietf-mls-combiner/02/ Work in progress.