POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea

Post-Quantum solutions for security protocols

Relators
prof. Danilo Bazzanella
dott. Maria Chiara Molteni
firma dei relatori

. . . .
. . . .

Candidate
Giacomo Greco
firma del candidato

. . . .

Academic Year 2023-2024

To my family.

To my grandparents.

Abstract

In an era where data security is an indispensable priority, the evolution of security protocols is crucial to ensure the protection of digital communications. This study focuses on integrating post-quantum solutions into the Transport Layer Security (TLS) security protocol to mitigate vulnerabilities associated with the advent of quantum computers that could lead to the break of our security protocols.

The TLS protocol represents a critical standard for ensuring confidentiality and integrity of communications over the Internet. However, its robustness has been questioned by the increasing computing power of quantum computers, which could compromise the cryptographic algorithms traditionally used by TLS.

This research work proposes an innovative modification of the TLS handshake algorithm, introducing post-quantum cryptography mechanisms to ensure enhanced security in the era of quantum computers. Implementing such post-quantum solutions within the TLS protocol offers robust protection against cryptographic attacks based on quantum algorithms.

Through a detailed analysis of performance and security, this thesis demonstrates the effectiveness of post-quantum solutions in increasing the resilience of the TLS protocol against advanced threats. The results obtained provide a solid start for integrating these emerging technologies into cybersecurity systems, while also ensuring the protection of sensitive data in an increasingly complex and threatening digital landscape.

Acknowledgements

Un ringraziamento va al mio relatore, il Professor Danilo Bazzanella, per la sua guida e i suoi consigli durante la stesura di questa tesi. Desidero ringraziare l'azienda Security Pattern SRL e il mio tutor aziendale Maria Chiara Molteni, per avermi dato l'opportunità di svolgere uno stage formativo e per la disponibilità e pazienza dimostrate durante tutto il periodo. La loro esperienza è stata fondamentale per arricchire questo lavoro.

Grazie alla mia famiglia: a mamma e papà, che mi hanno sempre incoraggiato e sostenuto, credendo in me anche nei momenti di difficoltà. Alle mie sorelle: Virna e Agnese per essere sempre state un punto di riferimento costante e insostituibile. Grazie per il vostro affetto e il vostro sostegno incondizionato.

Un ringraziamento dal profondo del cuore va ai miei amici più cari, Giulio, Luca ed Enrico. Con voi ho trovato non solo il supporto di cui avevo bisogno nei momenti difficili, ma anche la leggerezza che mi ha permesso di staccare la mente e ricaricare le energie. Che fossero chiacchierate profonde o semplici risate, ogni momento trascorso insieme ha reso questo viaggio unico e indimenticabile. Siete stati e siete tuttora il mio porto sicuro. Vi ringrazio per la vostra presenza costante, per non avermi mai fatto mancare il vostro affetto e il vostro incoraggiamento, e per avermi regalato i ricordi più belli che potessi desiderare. La nostra amicizia è una forza inarrestabile, una certezza che mi ha accompagnato in ogni passo di questo percorso, non esistono parole abbastanza forti per esprimere quanto vi sono grato. A voi devo l'adolescenza più felice che potessi sognare. Grazie di cuore, per tutto ciò che siete e per tutto quello che continueremo a essere insieme.

Grazie a Federico, conosciuto per caso durante questo viaggio universitario, ma che è diventato presto una presenza fondamentale nella mia vita. Abbiamo condiviso gran parte di questo percorso insieme, affrontando esami, progetti e infinite ore di studio, ma anche momenti di svago che hanno reso tutto più leggero. La tua amicizia ha reso questo cammino più sopportabile e molto più piacevole.

Un pensiero speciale va anche a Gli amici di Maicol, con i quali ho condiviso non solo tantissime serate indimenticabili, ma anche legami che vanno ben oltre il divertimento. Grazie per aver riempito di allegria e risate i miei giorni, e per essere una costante fonte di spensieratezza.

Grazie Giulia, ogni volta che le sfide sembravano insormontabili, la tua fiducia in me ha riacceso la mia determinazione. Ti sono grato per aver sempre creduto in me, per tutti gli alloggi a Roma, per le immagini di questa tesi e per tutto l'affetto che mi dai ogni giorno. Questo traguardo senza te al mio fianco, non avrebbe lo stesso sapore.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In today's digital era, data security is an indispensable priority, with security protocols playing a crucial role in protecting digital communications. As technological advancements continue, a new challenge emerges: the advent of post quantum computing.

Quantum computers, with their superior processing power, pose a significant threat to current cryptographic algorithms, potentially rendering some of them obsolete.

Many of the currently used security algorithms and protocols are vulnerable to these emerging threats. An example is The Transport Layer Security (TLS) protocol, which is a cornerstone for ensuring the confidentiality and integrity of Internet communications. Quantum algorithms pose a threat to the traditional cryptographic methods that form the basis of most security protocols, making it essential to develop and integrate post-quantum cryptographic solutions to protect digital communications.

This thesis explores the integration of post-quantum cryptographic mechanisms into the TLS protocol. The focus is to propose a modifications to the TLS handshake algorithm to incorporate post-quantum solutions, thereby enhancing the security and resilience of TLS against quantum computing threats. This research aims to demonstrate the viability and effectiveness of post-quantum solutions in fortifying TLS.

The structure of this thesis is as follows:

1. Chapter 2 delves into the fundamentals of cryptography, explaining how the main operations works such as: encryption, decryption, hash functions and the main symmetric and asymmetric algorithms; laying the groundwork for understanding the cryptographic challenges posed

by quantum computing.

2. Chapter 3 examines existing security protocols providing a detailed explanation of it, highlighting the parts where they can be vulnerable to a post-quantum attack.

3. Chapter 4 introduces post-quantum cryptography, explaining lattice-based and hash-based cryptographic algorithms. In this chapter the main post-quantum algorithms standardized by the NIST (CRYSTALS-Kyber, CRYSTALS-Dilithium, FALCON and SPHINCS+) are discussed, describing for each one the key generation process, the encryption/decryption mechanism for the encryption algorithms and the digital signature generation and verification for the signature ones.

4. Chapter 5 presents the proposed modifications to the TLS handshake protocols, which aims to make it post-quantum resistant. The chapter starts with the proposed modifications to the structures around the TLS protocol (such as the X.509 Certificate), and continue with the proposed changes to the structure of the TLS Handshake protocol to make it functional in a post quantum environment.

The thesis aims to contribute to the field of cybersecurity by providing an exhaustive state-of-the-art description for both the actual and the post-quantum solutions, paving the way for a secure digital future in the age of quantum computing.

Chapter 2

Cryptography

In today's interconnected world, where sensitive information is transmitted across vast networks, the need for secure communication has never been more critical. This chapter explain the main cryptography principles, mandatory for a better understanding of the security protocols used nowadays. From ancient methods of secret writing to the sophisticated algorithms powering today's secure communication protocols, cryptography has evolved over millennia to meet the ever-growing demands of privacy and security in the digital age.

Cryptography is the discipline that embodies the principles, means, and methods for the transformation of data in order to hide their semantic content, prevent their unauthorized use, or prevent their undetected modification. Cryptography has a rich history dating back thousands of years, with ancient civilizations developing rudimentary methods of encrypting messages to protect sensitive information like the **Caesar's cipher**, one of the simplest and earliest known encryption techniques. It is a type of substitution cipher where each letter in the plaintext is shifted a certain number of places down or up the alphabet, this fixed amount is the key. For example, if the key is 3, each letter in the message would be shifted three positions to the right in the alphabet. So, "A" would become "D," "B" would become "E," and so on. If the shift extends beyond the end of the alphabet, it wraps around to the beginning, so "X" would become "A," "Y" would become "B," and "Z" would become "C."

In the mid-1970s with the advent of computers, electronic communications begin to replace the printed paper in a large number of applications like communications between many people or many computers in different

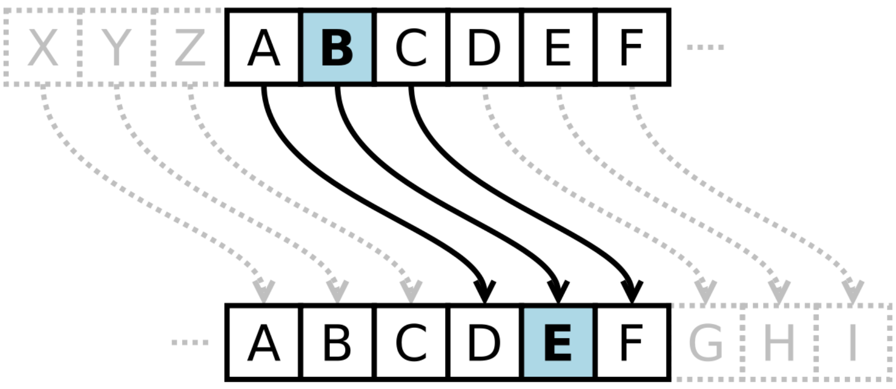


Figure 2.1: **Caesar's Cipher representation from** [16]

parts of the world creating different systems. Although different systems have different security goals, there are some generic goals:

Confidentiality: A system satisfies the confidentiality goal, if it prevents an attacker from disclosing some information defined as confidential. The main mechanism to obtain it is by performing encryption, with two main variants: symmetric encryption, and asymmetric encryption. This part will be more developed in the next section.

Integrity: the integrity property, as reported in the article [8] guarantee that data has not been altered in an unauthorized manner since it was created, transmitted, or stored .The main way to obtain it is by using hash functions and Message Authentication Codes (MAC) that are going to be explained in chapter 2.4.

Authenticity : the authenticity property, as reported in the article [26], tell us if data are originated from its presumed source. Usually authenticity is granted by the use of a digital signature, as explained in chapter 2.5.4

Non-Repudiation: is the property that provides evidence of the origin of information, that can be presented, later, to a third party, to 'prove' the identity of the origin. Usually also granted by signature schemes.

2.1 Mathematical Fundamentals

Before explaining the various parts by which cryptography is composed, a robust grasp of mathematical principles is needed as the incipit for the analysis of cryptographic algorithms and security protocols. In this section we will analyze the mathematical fundamental needed for a better understanding of the following chapters.

2.1.1 The AND operation

The AND operation, often denoted by the symbol " $\wedge$ ", is a fundamental logical operation in Boolean algebra and digital logic. It is commonly used to combine or compare the values of two binary variables according to the following truth table:

Input A	**Input B**	**Output**
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.1: AND truth table

the AND operation produces 1 only if both the input values are 1, otherwise it will produce 0.

2.1.2 The XOR operation

The XOR (exclusive OR) operation is a fundamental bitwise operation that plays a crucial role in many cryptographic algorithms and protocols. The XOR operation takes two binary inputs and outputs a single binary value according to the following truth table:

Input A	**Input B**	**Output**
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.2: XOR truth table

The peculiarity of this truth table is that it has the 50% of 0 and the 50% of 1. If XOR is performed with 2 random inputs then also the output will be equally random. XOR does not change the probability distribution of the input although it generates different outputs. For that reason is often used for its property of unpredictability, making it suitable for a variety of cryptgraphic operations as a "confusion" operation [13].

2.1.3 Modular Arithmetic

Modular arithmetic is a system of arithmetic for integers, where numbers "wrap around" when reaching a certain value, called the modulus. In modular arithmetic, the notation $amod(n)$ denotes the remainder when a is divided by m. For example, $8mod3 = 2$ because 8 divided by 3 leaves a reminder of 2. Modular arithmetic allows cryptographic operations to be performed efficiently with large integers by working with remainders rather than the full integer values. Many cryptographic primitives rely on modular arithmetic for their security properties such ad RSA and DSA, also modular arithmetic forms the basis of algebric structure: finite fields.

2.1.4 Finite Fields

A finite field is a field that contains a finite number of elements. Finite fields have several important properties:

1. **Closure**: The sum and product of any two elements in a finite field result in another element in the field.

2. **Associativity and Commutativity**: Addition and multiplication are associative and commutative in a finite field.

3. **Existence of Identity Elements**: A finite field has additive and multiplicative identity elements (0 and 1, respectively).

4. **Existence of Inverses**: : Every nonzero element in a finite field has a unique additive and multiplicative inverse.

. The most common examples of finite fields are given by the integers $modp$ when p is a prime number. The number of elements of a finite field is called its **order**.

2.1.5 Elliptic Curve

Elliptic curves are algebraic curves defined by equations of the form: $y^2 = x^3 + ax + b$ where a and b are constants, and the curve is defined over a finite field K. One of the most important properties of elliptic curves is their group structure, defined by the following rules:

1. **Addition**: Given two points P and Q on the curve, their sum is the reflection of the third point of intersection of the curve with the line passing through P and Q across the x-axis
2. **Identity Element**: The point at infinity serves as the identity element for addition. Adding the point at infinity to any point P results in P itself.
3. **Inverse**: For any point P on the curve, its inverse P is the point obtained by reflecting P across the x-axis.

2.1.6 Permutation

A permutation is one of the ways in which a number of things can be ordered or arranged. For example, the six permutations of the set 1, 2, 3 are: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), and (3, 2, 1). Permutations are used in almost every branch of mathematics and in many other fields of science.

2.1.7 Lattices

In geometry and group theory, a lattice in the real coordinate space $\mathbb{R}^n$ is an infinite set of points in this space with the properties that coordinate addition or subtraction of two points in the lattice produces another lattice point, that the lattice points are all separated by some minimum distance, and that every point in the space is within some maximum distance of a lattice point. An example of a lattice in the Euclidean plane is shown in figure 2.2

2.1.8 Basis

In mathematics, a set of vectors in a vector space V is called a basis if every element of V may be written in a unique way as a finite combination of elements of B.

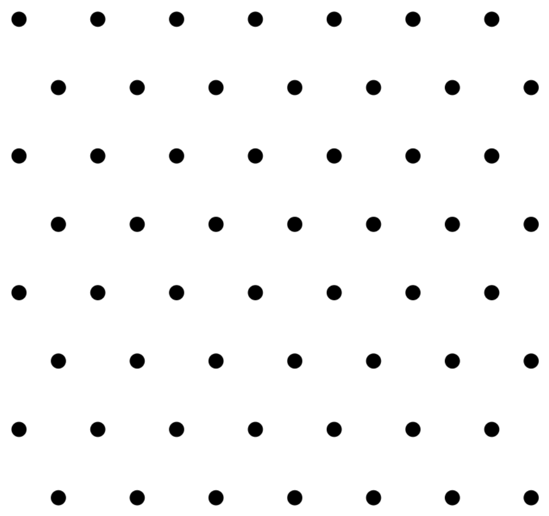


Figure 2.2: A Lattice in Euclidean Plane

As we can see in figure 2.3 the same Vector can be represented using two different bases that are the purple and red arrows.

2.1.9 Vector Space

A vector space (also called a linear space) is a set whose vectors, may be added together and multiplied by numbers called scalars. Vector spaces are characterized by their dimension, which, specifies the number of independent directions in the space.

2.1.10 Norm

In mathematics, a norm is a function from a real or complex vector space to the non-negative real numbers.

2.1.11 Learning with Error problem

In cryptography, learning with errors (LWE) is a mathematical problem that is widely used to create secure encryption algorithms. It is based on the idea of representing secret information as a set of equations with errors. In other words, LWE is a way to hide the value of a secret by introducing noise to it.

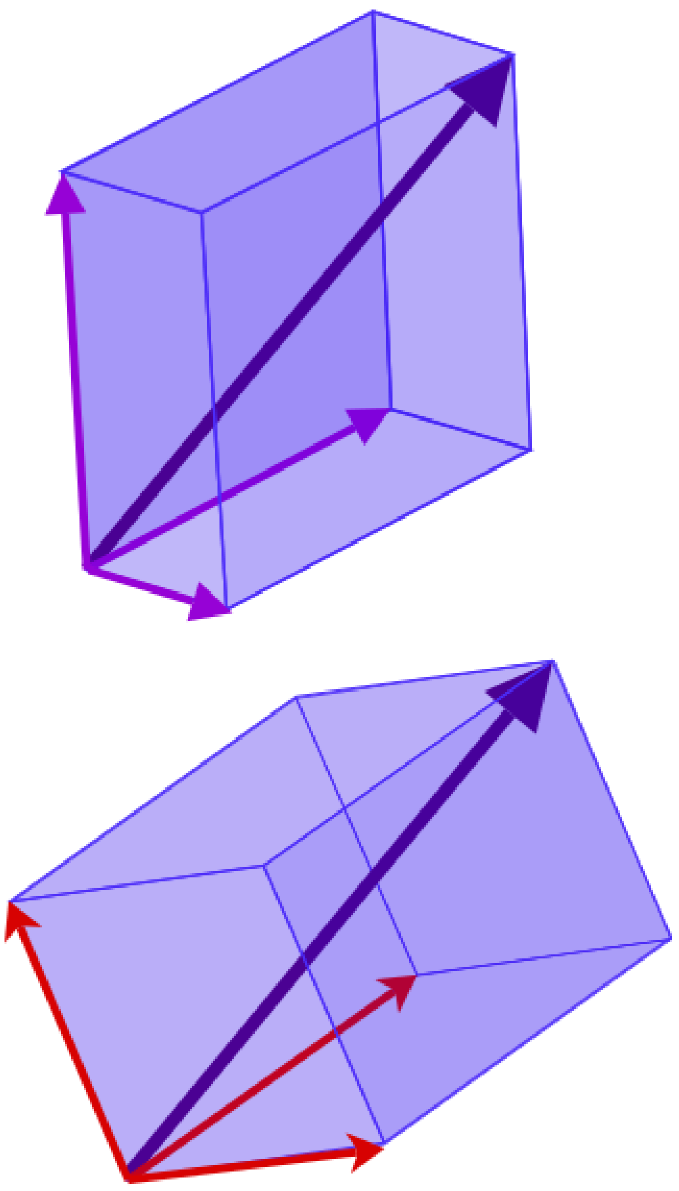


Figure 2.3: The same Vector can be represented by using different bases. Picture from [35]

2.1.12 Closest Vector Problem

The Closest Vector Problem (CVP) involves finding the closest lattice vector to a given point. Given a lattice L defined by a basis B, and a target vector t not necessarily in the lattice, the goal is to find a lattice vector $v \in L$ that minimizes the distance $\| v - t \|$.

2.1.13 Indistinguishability under chosen plaintext attack

Indistinguishability under Chosen Plaintext Attack (IND-CPA) is a security property of encryption schemes. It ensures that an attacker, who can choose plaintexts and obtain their corresponding ciphertexts, cannot distinguish between the encryptions of any two chosen plaintexts with a probability significantly better than random guessing.

2.1.14 Indistinguishability under non-adaptive and adaptive Chosen Ciphertext Attack

IND-CCA1 is a security property of encryption schemes ensuring that an attacker, who can choose ciphertexts to be decrypted (but only before seeing the challenge ciphertext), cannot distinguish between the encryptions of any two chosen plaintexts.

2.2 Encryption

Encryption serves as a fundamental technique in the domain of cybersecurity, providing a way to secure sensitive data by transforming it into an unintelligible form that can only be deciphered by authorized parties, having a crucial role in ensuring the confidentiality property of a system. Encryption is done by using mathematical techniques which performs the transformation of the messages we want to send by using **encryption** algorithms and **decryption** algorithms. For that reason a sender that is encrypting the message can be confident that only the intended receiver will have the possession of the right key to decrypt the original text, for that reason the encryption operation permits to achieve the confidentiality property. We can see in figure 2.4 how the basic encryption works.

The common terminology used in cryptography includes two keywords

1. **Plaintext**: the message in clear, that it has not been modified yet.

2. **Ciphertext**: the encrypted message.

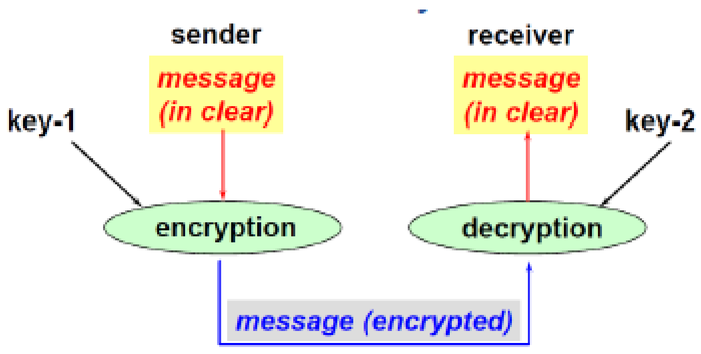


Figure 2.4: Basic encryption

The encryption algorithms takes a message and transform it in a way which is no more understandable using an encryption algorithm with a secret key (key-1 in figure 2.4), generating a Ciphertext. To recover the original message, the decryption algorithm will make the message readable again only by using the same algorithm in decryption mode and the same key used for the encryption or a different one(Key-2 in figure 2.4, assuming the two key are different).

Depending on which relation exists between key-1 and key-2 there are two different kinds of cryptography: if sender and receiver are using only a single shared key we are in the **Symmetric Cryptography** field, otherwise if sender and receiver are using two separate keys we are in the **Asymmetric Cryptography** field. Moreover symmetric encryption algorithms are divided into two categories based on the input type: **Block cipher** is an encryption algorithm that takes a fixed size of input and produces a ciphertext of the same size of the input and **Stream cipher** if the algorithm does not works on fixed block.

2.3 Symmetric Cryptography

Symmetric cryptography, known also as secret key cryptography, is a cornerstone of modern cryptographic systems, plays a fundamental role in securing data through the use of shared secret keys only known to the communicating parties, to both encrypt and decrypt messages. In figure 2.5 we have an high level schema on how it works.
We can see a plaintext used as an input for the E (Encryption) block that takes also the Key ad an input. The result is a non-understandable text

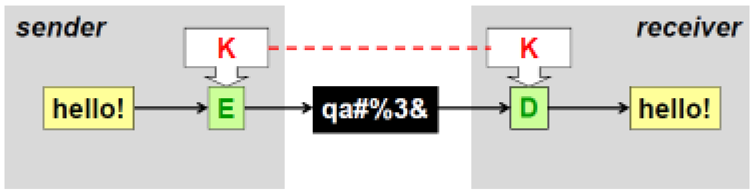


Figure 2.5: **Symmetric Cryptography**

that is sent to the receiver. To recover the original text the D(Decryption) block algorithm is used with the **same** key used to encrypt the original text, if a different key is used the output will still be available but remains non-understandable.

2.3.1 Symmetric Cryptography Algorithms

This chapter explores the principles, functionalities, and applications of the main symmetric cryptography algorithms such as DES, TripleDES and AES.

DES

DES stands for Data Encryption Standard, a historic encryption algorithm representing one of the earliest and most widely recognized encryption algorithms. Initially developed by IBM in the early 1970s, DES was adopted as a federal standard for encryption by the United States government in 1977. DES was widely published and openly available for scrutiny and analysis by the cryptographic community but the specifications did not include details about the algorithm's S-Box that was viewed as proprietary to IBM. However, these details were eventually made public, and DES was subject to extensive cryptanalysis over the years. Despite its widespread use and standardization, concerns eventually arose regarding the security of DES, particularly due to its relatively short key length of 56 bits. Over time, advancements in computing power rendered brute-force attacks against DES feasible, leading to its eventual deprecation as a standard encryption algorithm in favor of more secure alternatives like TripleDES and the Advanced Encryption Standard (AES).

DES is a block cipher which encrypts data in blocks of size of 64 bits each, that means 64 bits of plaintext go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and decryption. Here is an explanation on how DES works based on the research in [62] and on the original paper [29]. We have mentioned that DES uses a 56-bit key. Actually, The initial key consists of 64 bits. However, before the DES process even starts, every 8th bit of the key is discarded (for checking parity) to produce a 56-bit key. Parity bits are a form of simple error detection where, for each byte, a bit is used to ensure that the total number of 1-bits in the string is even or odd. Indeed bit positions 8, 16, 24, 32, 40, 48, 56, and 64 are discarded. Thus, the discarding of every 8th bit of the key produces a 56-bit key from the original 64-bit key. DES consists of 16 steps, each of which is called a round and each round performs the **Feistel schemes** operations.

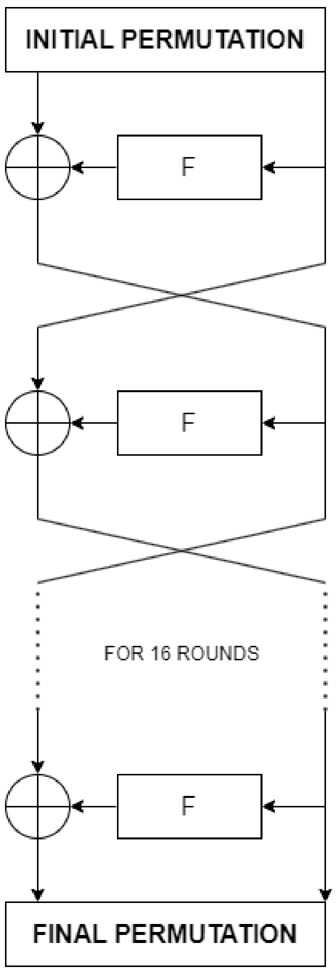
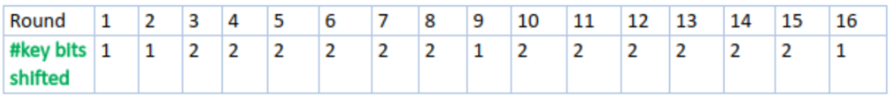


Figure 2.6: **DES algorithm**

The algorithm's overall structure is shown in figure 2.6: there are 16 identical stages of processing, called rounds. Before the main rounds, the block is divided into two 32-bit halves and processed alternately, this process is known as the Feistel scheme (referred ad F in figure 2.6). The Feistel structure ensures that decryption and encryption are very similar processes—the only difference is that the keys are applied in the reverse order when decrypting. The rest of the algorithm is identical. This greatly simplifies implementation, particularly in hardware, as there is no need for separate encryption and decryption algorithms. The F-function operates on half a block (32 bits) at a time and consists of the following stages: Key Transformation, Expansion Permutation, S-box permutation and P-box Permutation

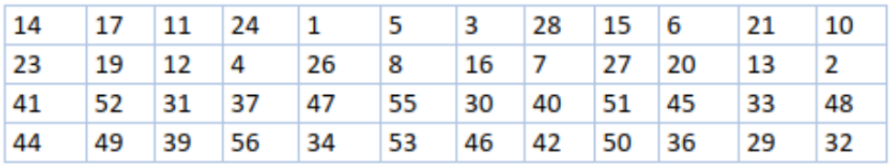
Key Transformation As we previously said a 64-bit key is transformed into a 56-bit key by discarding every 8th bit of the initial key, from this 56-bit key, a different 48-bit Sub Key is generated during each round. The 56 bit key is divided in two 28 bits half that are circularly shifted by one or two position depending on the round.



Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
#key bits shifted	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Figure 2.7: Number of keys shifted per rounds

We can use figure 2.7 for a better understanding, if the round numbers 1, 2, 9, or 16 the shift is done by only one position for other rounds, the circular shift is done by two positions and so on. After an appropriate shift, 48 of the 56 bits are selected.



14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

Figure 2.8: Compression Permutation

For instance as shown in figure 2.8, after the shift, bit number 14 moves to the first position, bit number 17 moves to the second position, and so on. Observing the table, we will realize that it contains only 48-bit positions. Bit number 18 is discarded (we will not find it in the table), like 7 others, to reduce a 56-bit key to a 48-bit key. Since the key transformation process involves permutation as well as a selection of a 48-bit subset of the original 56-bit key it is called Compression Permutation. This compression permutation technique permits to use a different subset of the key bits in each round.

Expansion Permutation In the Expansion Permutation the 32-bit half-block is expanded to 48 bits. This happens as the 32-bit block is divided into 8 blocks, with each block consisting of 4 bits. Then, each 4-bit block of the previous step is then expanded to a corresponding 6-bit block resulting in 2 more bits added. Now the 48-bit key is XOR with 48-bit of the precedent block and the 48 bit of the other half pf the block and the resulting output is given to the next step, which is the S-Box substitution.

S-box Permutation The S-box (substitution box) permutation is a procedure that accepts the 48-bit input from the XOR operation containing the compressed key and expanded half block and creates a 32-bit output utilizing the substitution technique.The substitution is implemented by the eight substitution boxes (also known as the S-boxes). Each 8-S-boxes has a 6-bit input and a 4-bit output. The 48-bit input block is divided into 8 sub-blocks (each including 6 bits), and each sub-blocks is provided to an S-box.

The substitution in each box follows a pre-decided rule depends on a 4-row by 16- column table. The sequence of bits one and six of the input represent four rows and the sequence of bits two through five represent sixteen columns.

Because each S-box has its own table, we require eight tables, as display in table 1 to table 8, to represent the output of these boxes. The values of the inputs (row number and column number) and the values of the outputs are given as decimal numbers to store space as explained in article [22].

P-box Permutation In this step the new resulting blocks are the input for 2 permutation box so their bits are transposed again, similar to the step discussed in figure 2.7. This p-box helps in adding confusion and diffusion, so as to make the attacker difficult to decrypt the message hidden. The input block bits are permutated and hence the confidentiality of the message is preserved. As explained in [46] this process helps in securing the message more efficiently. In the end, the two previously halved block are rejoined and a Final Permutation is performed on the combined block. The result of this process produces 64-bit ciphertext.

Before DES was adopted as a national standard, the creators of public key cryptography, Martin Hellman and Whitfield Diffie, registered some objections to the use of DES as an encryption algorithm fearing an attack by an intelligence association [24].

Diffie and Hellman then outlined also a **brute force attack**[11] on DES and in 1998, under the direction of John Gilmore of the EFF, a team spent 220,000 and built a machine that can go through the entire 56-bit DES key space in an average of 4.5 days. On July 17, 1998, they announced they had cracked a 56-bit key in 56 hours. For that reasons the algorithm was retired in 2005. [24]

Triple DES

Triple DES is an encryption algorithm based on the original Data Encryption Standard (DES). It is a symmetric encryption algorithm that uses multiple rounds of the Data Encryption Standard (DES) to improve security in particular it is known as Triple DES because it uses the Data Encryption Standard (DES) three times to encrypt its data.As said in article [30] it outperforms the original Data Encryption Standard (DES). The idea behind using three DES passes instead of two is designed to prevent **Man in the Middle attacks**. In a setting with Double-DES implementation, an adversary could read the encrypted output of the DES passes at each end of the stream. They could then use a brute-force attack to find relationships between the block set and the mathematical operator. Keys that produce the same result in either direction could also be identified as possible solutions to the DES implementation. According to NIST [17] the Triple DES scheme that uses three different keys offers a 100-bit security level which is considered acceptable until the year 2030.

Figure 2.9 describe how 3DES works: the encryption scheme is $C(x) = E_{K3}(D_{K2}(E_{K1}(P(x))))$ where C stand for Ciphertext, E for Encryption, D for Decryption and P for plaintext, that means encrypt the plaintext using key K1; decrypt using key K2 and encrypt the resultant using K3 while the Decryption scheme can be denoted as: $P(x) = D_{K3}(E_{K2}(D_{K1}(P(x))))$

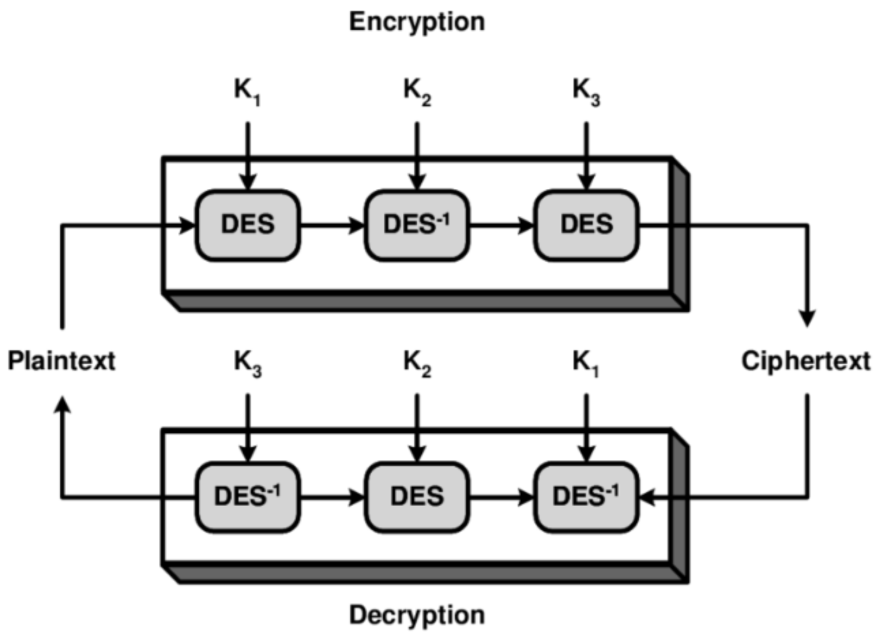


Figure 2.9: 3Des Scheme by [32]

AES

AES stands for Advanced Encryption Standard and is the most used symmetric algorithm nowdays. AES supports key lenghts of 128,192 and 256 bits and a longer key means a higher level of security. AES has the ability to deal with 128 bits (16 bytes) as a fixed plaintext block size, these 16 bytes are represented in 4x4 matrix. The core of the algorithm is a sequence of fixed transformations of the state called rounds, and there are a prefixed number of rounds based on the key lenght (10 rounds for the 128 bits key, 12 for the 192 bits key and 14 for the 256 bits one).

As we can see in figure 2.10 a round on encryption can be decomposed in four transformations: **Substitute Bytes, Shift Rows, Mix Columns, Add Round Key**, for a better understanding of the algorithm we refer to the FIPS 197 document available on [45].

Substitute Bytes Transformation The first stage of each round starts with SubBytes transformation. This stage depends on nonlinear S-box to substitute a byte in the state to another byte.

For a better visualization we can use 2.3: if in the state we have "AC", it

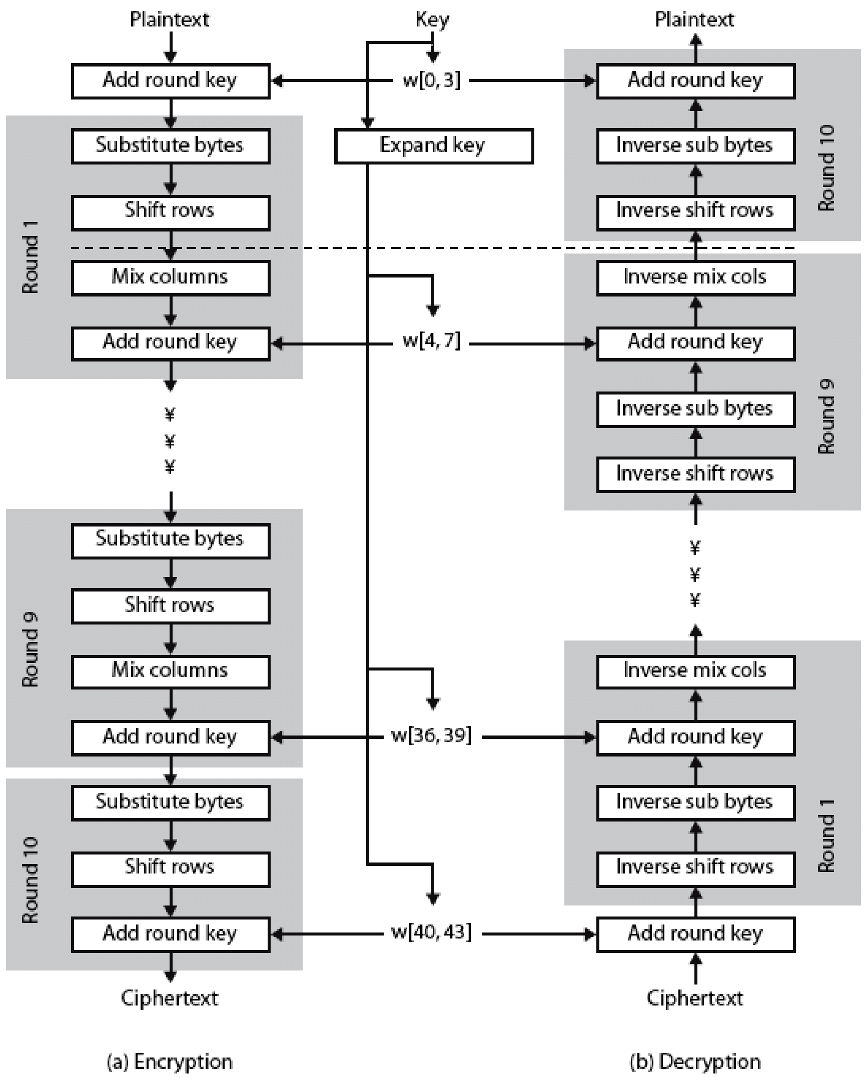


Figure 2.10: Basic structure of AES Algorithm by [2]

has to replace to 7 4, created from the intersection of A and C.

Shift Rows Shift Row is a transformation of the state in which the bytes in the last three rows of the state are cyclically shifted to the left in each row rather than the first row.

In picture 2.11 we can visualize the transformations done to the matrix: row zero remains fixed and does not carry out any permutation. In the first row only one byte is shifted circularly to the left. The second row is shifted

Table 2.3: AES S-box Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	B2	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	DB	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	3C	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	CB	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	6B	41	99	2D	0F	B0	54	BB	16

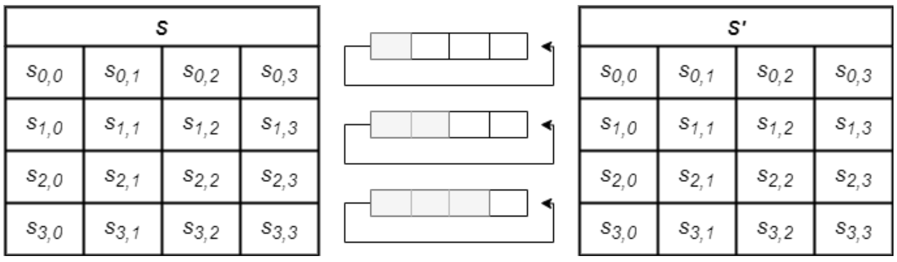


Figure 2.11: AES Shift Rows

two bytes to the left. The last row is shifted three bytes to the left.

Mix Columns Mix Columns is a transformation that multiplies each of the four columns of the state by a single fixed matrix. In another word, each row of matrix transformation must multiply by each column of the state. The results of these multiplication are used with XOR to produce a new four bytes for the next state.

Add Round Key AddRoundKey is the most important stage in AES algorithm and it is a bitwise XOR operation that combines the current ciphertext being encrypted with a portion of the encryption key. Both the key and the input data are structured in a 4x4 matrix of bytes as shown in figure

2.11. Before encryption begins, the original encryption key is expanded into multiple round keys, and each round key is derived from the original key and is used in a specific round of the AES encryption process. At this point each byte of the state matrix is XORed with the corresponding byte of the round key. In figure 2.12 we have a visual representation on the transformation.

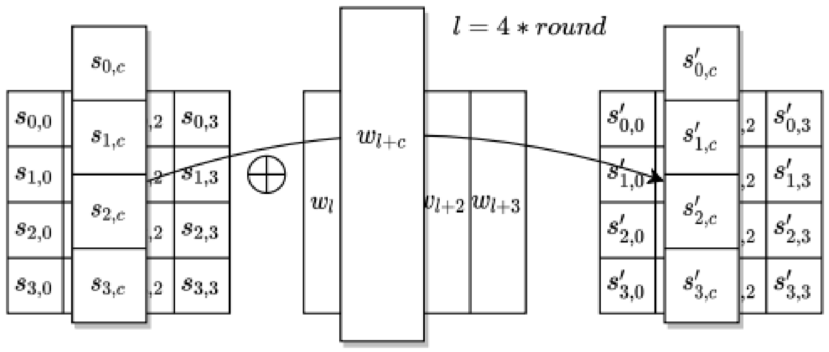


Figure 2.12: Add Rund Key transformation schema by [45]

The AddRoundKey transformation introduces the key material into the data being encrypted, effectively mixing the original data with the encryption key.

2.3.2 Symmetric Algorithms application modes

Central to the implementation of symmetric algorithms are their application modes which delineate the specific techniques and procedures through which data is encrypted, decrypted, and processed.

Electronic Code Book

Electronic Code Book (ECB) is a simple mode of operation with a block cipher that's mostly used with symmetric key encryption. It is a straightforward way of processing a series of sequentially listed message blocks.

The input plaintext is broken into numerous blocks made by 128 bit each, and if the plaintext's bit are not multiple of 128 then some padding methods will be used. Paddings are mechanisms that allows to append some predefined values to a block permitting it to fulfill the size of a block. The blocks are individually and independently encrypted (ciphertext) using the encryption key. As a result, each encrypted block can also be decrypted

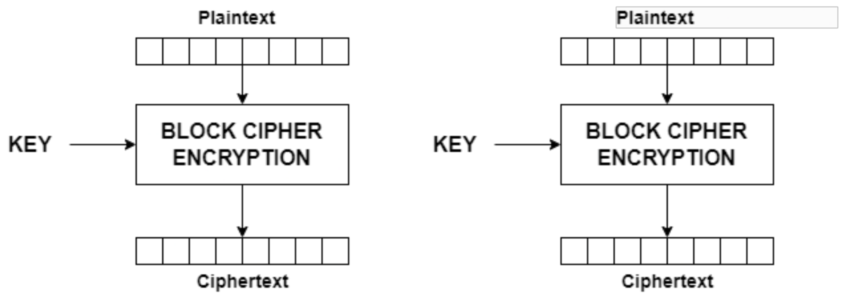


Figure 2.13: ECB encryption mode

individually. ECB can support a separate encryption key for each block type.

As said in [5],in ECB each block of plaintext has a defined corresponding ciphertext value, and vice versa. So, identical plaintexts with identical keys always encrypt to identical ciphertexts. This means that if plaintext blocks P1, P2 (with P1 and P2 i refer to figure 2.13 where P1 id the first block and P2 the second one) and so on are encrypted multiple times under the same key, the output ciphertext blocks will always be the same so, the same plaintext value will always result in the same ciphertext value. This also applies to plaintexts with partial identical portions. For instance, plaintexts containing identical headers of a letter and encrypted with the same key will have partially identical ciphertext portions. ECB is ideal for limited data volumes otherwise an attacker can exploit its regularity and decrypt the data.

Cipher Block Chain

Based on [14] Cipher Block Chaining (CBC) is a mode of operation in symmetric key block cipher cryptography, enhancing security by combining each plaintext block with the previous ciphertext block prior to encryption. This creates a chain-like dependency, where initial plaintext blocks influence the successive ones, making the encrypted output more randomly distributed.

In figure 2.14 we can see how CBC works.The practical application of Cipher Block Chaining involves integrating an initialization vector (IV) XORed with the first block of plaintext (P1) to add an additional layer of randomness, consequently boosting security by keeping encrypted data unpredictable. The encrypted output (C1) is then XORed with the next block of plaintext (P2) to modify the input of the current block before it

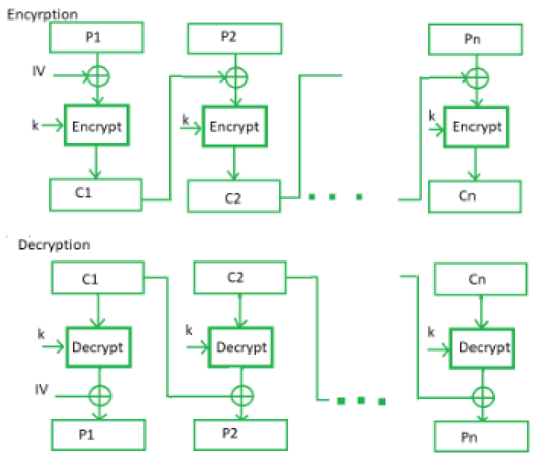


Figure 2.14: Cipher Block Chaining

is passed through the encryption algorithm, ultimately resulting in a chain of dependency between the data blocks. In this manner, even if identical plaintext blocks are encountered, the encrypted output will differ, rendering pattern analysis futile. CBC serves as a trusted method in bolstering cryptographic security, making it a widely adopted strategy for safeguarding sensitive data in various industries like finance, telecommunications, and e-commerce [14].

Counter mode

In the Counter (CTR) mode the next keystream block is obtained by encrypting successive values of a "counter" which can be any function which produces a sequence which is guaranteed not to repeat for a long time, although an actual increment-by-one counter is the simplest and most popular.

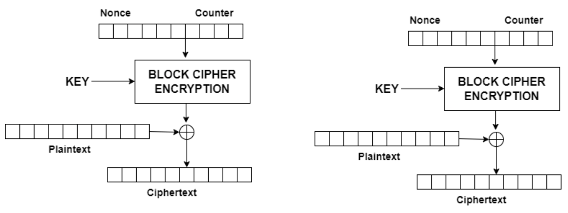


Figure 2.15: Counter Modes

If the nonce is random, then they can be combined with the counter using any invertible operation (concatenation, addition, or XOR) to produce the actual unique counter block for encryption. Counter mode uses an arbitrary number (the counter) that changes with each block of text encrypted. As shown in figure 2.15, the counter is encrypted with the cipher, and the result is XORed into ciphertext. Since the counter changes for each block, the problem of repeating ciphertext that results from the electronic code book (ECB) method is avoided.

Stream Cipher

Stream ciphers are a type of encryption algorithm that process an individual bit of plaintext at a time and does not need a fixed-size block. Stream ciphers are often faster than block ciphers in hardware and require circuitry that is less complex [21]. A stream cipher is a symmetric-key cipher where plaintext digits are combined with a pseudo-random cipher bitstream (keystream). Each plaintext digit is encrypted one at a time with the corresponding digit of the keystream, to give a digit of the ciphertext stream. Since encryption of each digit is dependent on the current state of the cipher, it is also known as state cipher. The combining operation is often a XOR.

The pseudo-random keystream is typically generated serially from a random seed value which serves as the cryptographic key for decrypting the ciphertext stream. Stream ciphers can be classified into **synchronous and self-synchronizing**.

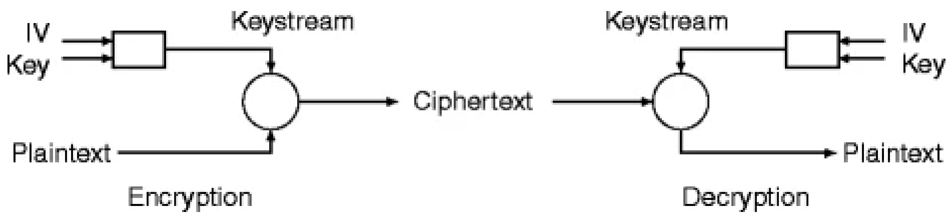


Figure 2.16: Synchronous Stream Cipher by [21]

Synchronous ciphers have an independently generated keystream from the plaintext and ciphertext. They need to use the same key in order to decrypt the data properly. If a ciphertext character is modified, it does not affect the decryption of the rest of the ciphertext, but if a character is deleted or inserted, synchronization will be lost and the rest of the decryption will fail.

As shown in figure 2.5 self-synchronizing (asynchronous) ciphers have a keystream that is generated from the key and a specified number of previous

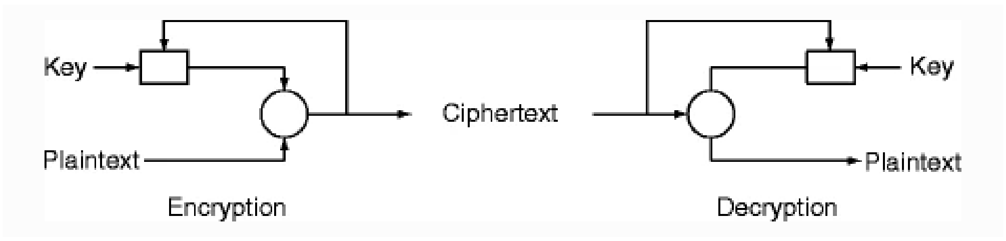


Figure 2.17: Self-Synchronizing Stream Cipher [21]

ciphertext characters. This type of stream cipher can better handle characters being deleted or inserted as the state only depends on the specified number of previous ciphertext characters. After that number has been processed, the cipher will be synchronized again. A synchronous stream cipher has no error propagation, but a self-synchronizing cipher has limited error propagation. If a character is modified, the maximum number of incorrect decryptions would be limited to the specified number of previous ciphertext characters after which correct decryption would resume.

2.4 Hash Functions

Hash functions are functions that compress an input of arbitrary length to a result with a fixed length and they are a very powerful tool to protect the authenticity of information. And in this chapter we are going to analyze them referring to [31] the main characteristics of an hash function are:

1. Deterministic: For a given input, a hash function always produces the same output.

2. Fixed Output Size: A hash function produces a fixed-size output, regardless of the input size.

3. Collision Resistance: It's computationally difficult to find two different inputs that produce the same hash output.

4. Pre-image Resistance: Given a hash output, it's computationally infeasible to find the original input.

5. Image-Resistance : It's computationally infeasible to find two different inputs that produce the same hash value.

6. Avalanche Effect: A small change in the input should result in a significantly different hash output.

7. Non-invertible: It's practically impossible to reverse the hash function to obtain the original input from the hash output.

Hash code works as a unique fingerprint identification digital control. If even a single bit of control changes, the hash code will change radically.

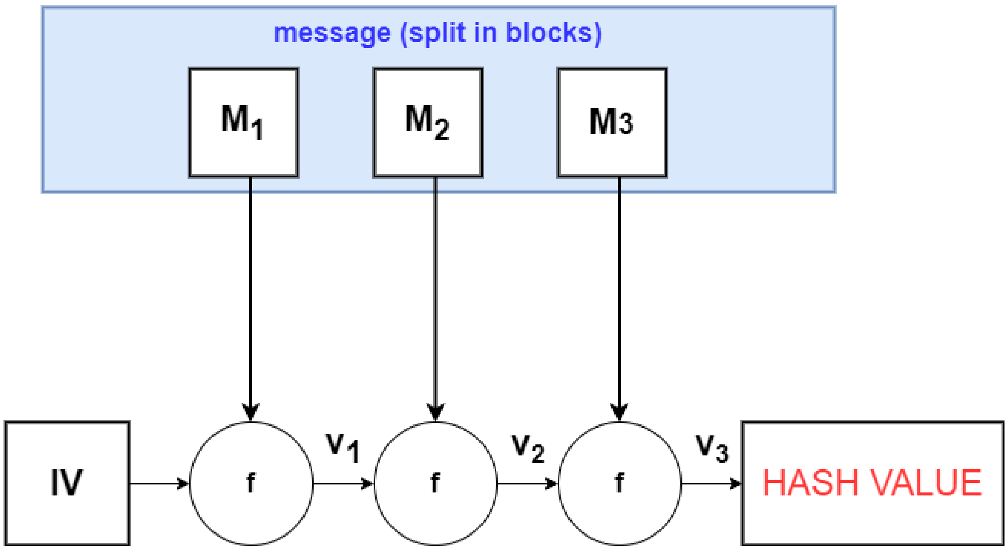


Figure 2.18: How a cryptographic hash function works.

In figure 2.18 we can visualize how an hash algorithm works:

1. split the message M in N blocks $M_1 ... M_N$

2. iteratively an hash function (f) will be applicated on each block producing a digest that is a unique representation of the original input

3. for each block a digest is calculated as follow :$V_k = f(V_{k-1}, M_k)$ with $V_0 = IV$ and $V_N = H$ where the IV is an initialization value which does not need to be announced or be random and H is the final hash calculated.

The most widely used hash algorithms nowadays are those ones in SHA (Secure Hash Algorithm) family published by the National Institute of Standards and Technology, with SHA-2 and SHA-3 being the most representative.

2.4.1 SHA-2 and SHA-3 algorithms

Name	Block Size	Digest Size
SHA-224	512 bit	224 bit
SHA-256	512 bit	256 bit
SHA-384	1024 bit	384 bit
SHA-512	1024 bit	512 bit

Table 2.4: SHA-2 algorithms

In table 2.4 we can see the algorithms in SHA-2 family, they have the same block size but an increasingly digest size. The digest lenght is important to avoid **aliasing** that is the collision of two different messages on the same digest. Breaking a digest algorithm means finding a second message that will generate the same digest of the first one breaking the **Collision Resistance property**, an algorithm that generates a digest of N bits, has a probability of aliasing (P_a) $P_a \sim 1/2^N$ so, for statistical reason an algorithm with many bits is preferable, right now SHA-256 is considered secure enough with a 256 bits digest.

We can refer to figure 2.19 for a visualization on how the SHA-2 hashing process works.

The first thing that's worth noting is that the diagram shows Round 0, Round T, and Round 63. Round 0 is the first round, while Round T is a placeholder that represents any round in between. Round 63 is the final round, which gives us a total of 64 rounds (note that SHA-384 and SHA-512 involves 80 rounds) we will continue the explanation assuming a 512 bit block size. The input message (called M in figure 2.19 is padded to ensure its length is a multiple of the block size used by the SHA-2 variant (multiple of 512 bit for SHA-224 and SHA-256, multiple of 1024 bits for SHA-384 and SHA-512). The cut-off point for splitting blocks is actually either 447 bits (or 895 bits), because we need at least one bit of padding, plus a 64-bit (or the 128-bit) message containing the length of the block must be included. At the end of this process we will obtain 16 blocks indicated in figure as $W_0 ... W_{63}$,this implies that we need another 48 values before we have all of our W inputs. The remaining inputs are calculated using the following formula: $W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$. t is a stand-in indicating in which round we are up to, $\sigma_1(x)$ stands for the following

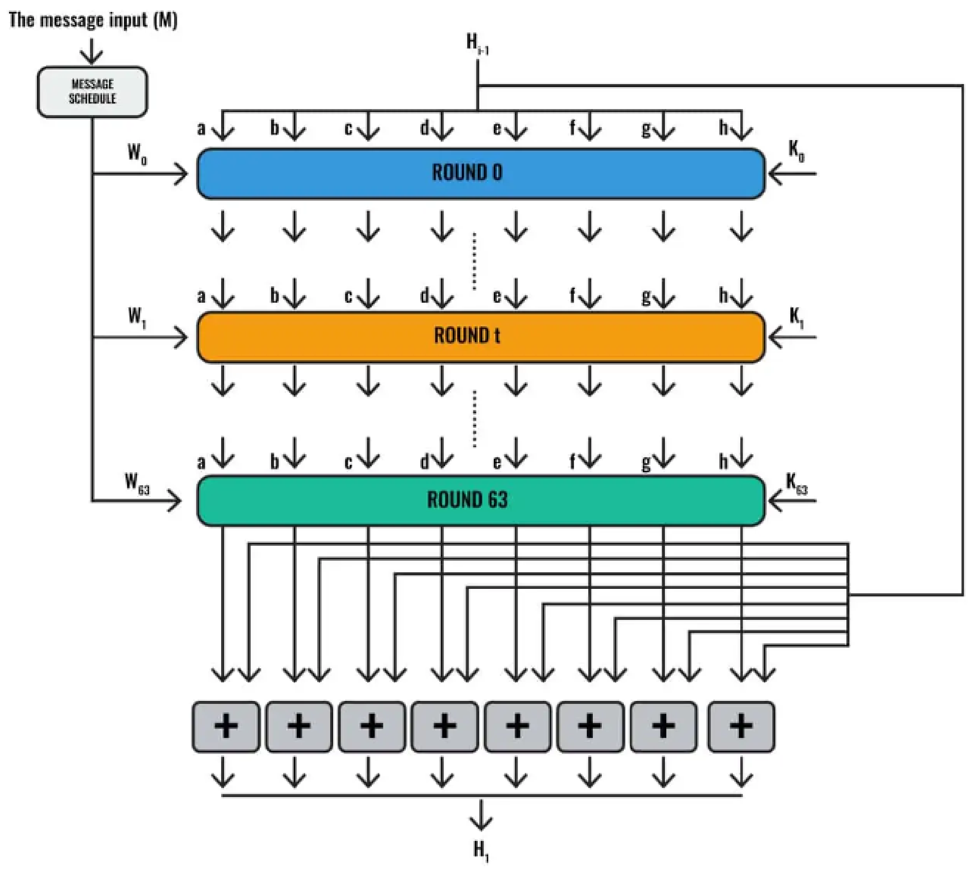


Figure 2.19: The SHA-2 Algorithm schema from [31]

equation: $\sigma_1(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$. $ROTR^{17}(x)$ is a circular right shift (for 17 bits in this case) to the value that follows it (in this case, x). $SHR^{10}(x)$ otherwise is a logical left-shift, so we shift to the left the value x by, in this case, 10 bits. $\oplus$ is the XOR operator. By applying this formula to the 16 W we already have we will obtain the remaining ones. At the top of the figure, $Hi - 1$ represents **the working variables**, which act as inputs in each round. There are eight of these variables, and they are updated at the end of each round. The first eight values are fixed and corresponds to the square root of the first eight prime numbers, in subsequent rounds these values will be different and indicated in the figure as a, b, c, d, e, f, g and h. In the right side of the rounds in the figure, there is another input, K. Those are 64 separate 32-bit values for K, one for each of the 64 rounds, derived from the cube-roots of the first 64 prime numbers.

After defining all the inputs we can refer to figure 2.20for a visualization of how SHA-2 uses all of them.

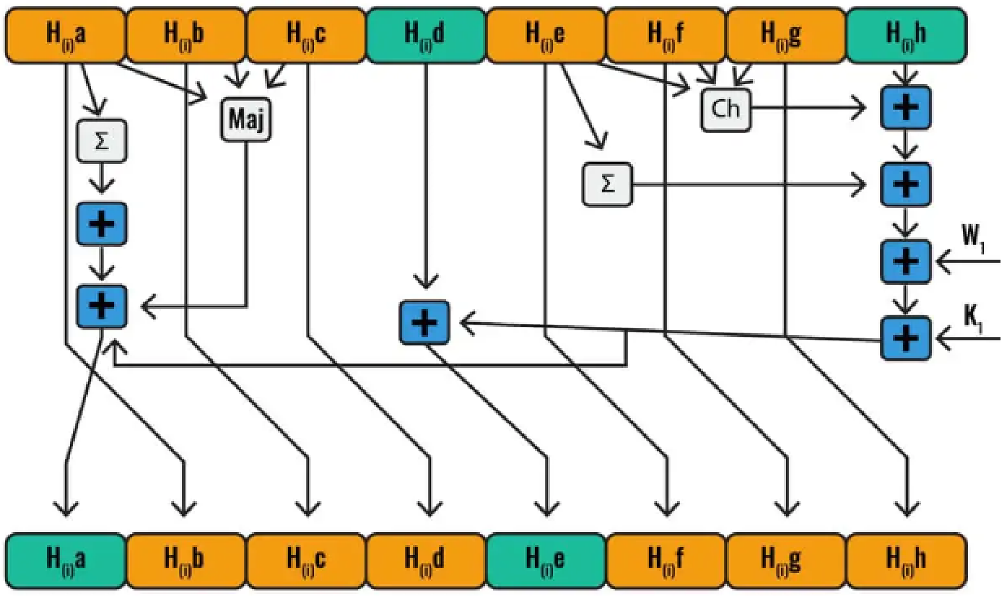


Figure 2.20: The SHA-2 calculations involved in a single round, from [31]

At the top we have the working variables (previously defined as a,b,c,d..h) and in the first round they will be the prefixed values cited before. The Maj box stands for the following equation: $Maj(a, b, c) = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$ where $\oplus$ is the XOR operation and $\wedge$ is the AND operator. The next block to be analyzed is the $\sum$ one, this block is translated into the following equation: $\sum_0(a) = ROTR^2(a) \oplus ROTR^{13}(a) \oplus ROTR^{22}(a)$. The blocks with the "+" symbol inside represents a modular addition of the inputs. The box labeled as "Ch" produces the following equation as an output: $ch(e, f, g) = (e \wedge f) \oplus (NOTe \wedge g)$. All of those operations are done in one round of the SHA-2 algorithm and after 63 rounds, we can see from figure 2.4 that the eight final H values are all input into a series of modular addiction boxes that gives as a result the SHA-2 hash desired.

As a consequence of statistical considerations (birthday paradox), a N-digest algorithm in insecure when more than $2^{n/2}$ digest are generated so, if an attacker sniffs the network, he can simply store all the messages and its digests passing by. With many messages and digests stored the probability to have the same digests for two different messages is high. At this point it is possible to exchange the two messages, because they have the same digest. Because of this problem the National Institute of Standards and technology initiated the SHA-3 competition in 2007 with a goal to develop a new standard for cryptographic hash functions that would complement existing algorithms like SHA-2 and provide an alternative in case they were compromised. The competition was won by the **Keccak algorithm** done by G.Bertoni, J.Daemen, G. Van Assche and M.Peeters announced by NIST in the special pubblication 800-185 [42]. The following part of the chapter will be an analysis of how the Keccak algorithm works based on the article [4]. Keccak is a hash function consisting of four cryptographic hash functions and two extendable-output functions. These six functions are based on an approach called **sponge functions**.Sponge functions provide a way to generalize cryptographic hash functions to more general functions with arbitrary-length outputs. The block diagram of SHA-3 consists of four functional blocks called state function, round constant, buffer function and Keccak function as shown in Figure 2.21.

The algorithm receives a matrix as input, called **state**. The state has a length of 1600 bits and consists of a three dimensional 5 x 5 $\times$ (word - sizetable), where *word-sizetable* $\in 1, 2, 4, 8, 16, 32, 64$. The buffer function has two input parameters, the first is the number of bits (64 bits or 256 bits) that accepts as input data, and the second is the state matrix and generates hash output based on sponge construction. Depending on the desired output length, the algorithm uses two parameters for the sponge construction as shown in Figure 2.22.

The two input parameters are the bitrate with r-bits, and the capacity with c-bits. The input message is padded and divided into blocks of r-bits. The function indicated as f is the main processing part and consists of 24 rounds. We are going to denote with b the width of the state that is made by r (the outer state) + c (the inner state). Also in the next formula we refer to the state matrix as A. The process of function f are theta ,rho ,pi ,chi and iota denoted as $\theta, \rho, \pi, \chi and \iota$ respectively.

1. θ consists of a parity computation, a rotation of one position, and a bitwise XOR as shown in the following equation:

$$\begin{aligned}
C[x] &= A[x,0] \oplus A[x,1] \oplus A[x,3] \oplus A[x,4] && 0 \leq x \leq 4 \\
D[x] &= C[x\text{-}1] \oplus ROT(C[x\text{+}1],1) && 0 \leq x \leq 4 \\
A[x,y] &= A[x,y] \oplus D[x] && 0 \leq x,y \leq 4
\end{aligned}$$

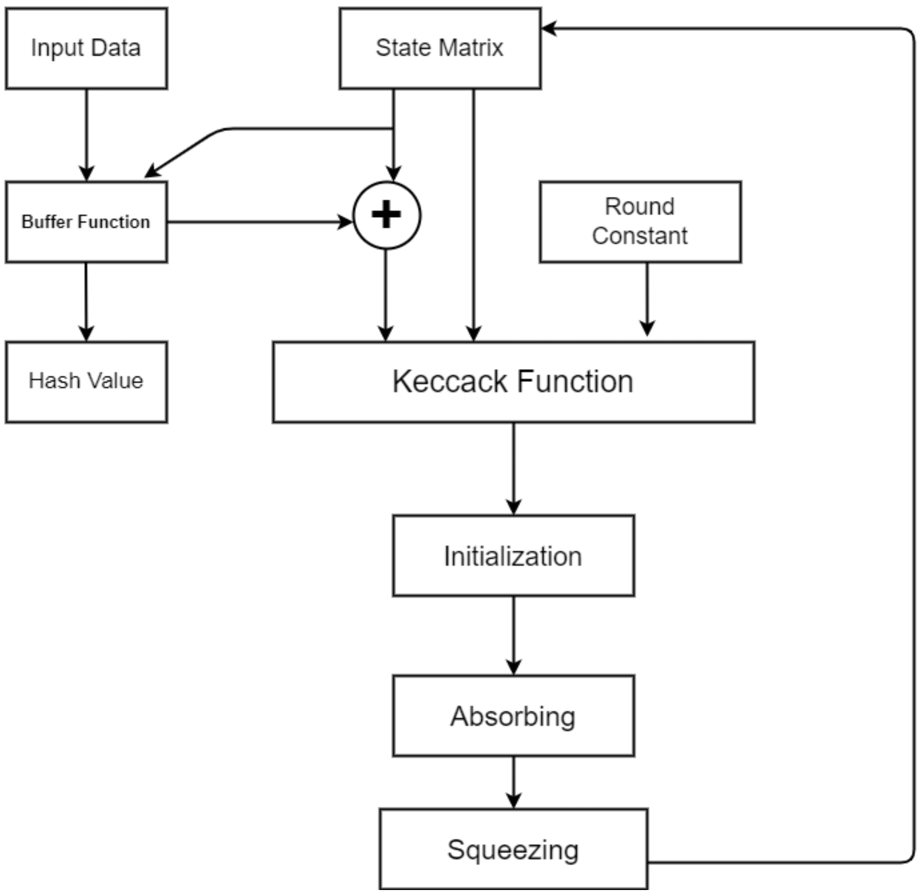


Figure 2.21: **Keccak Block Diagram**

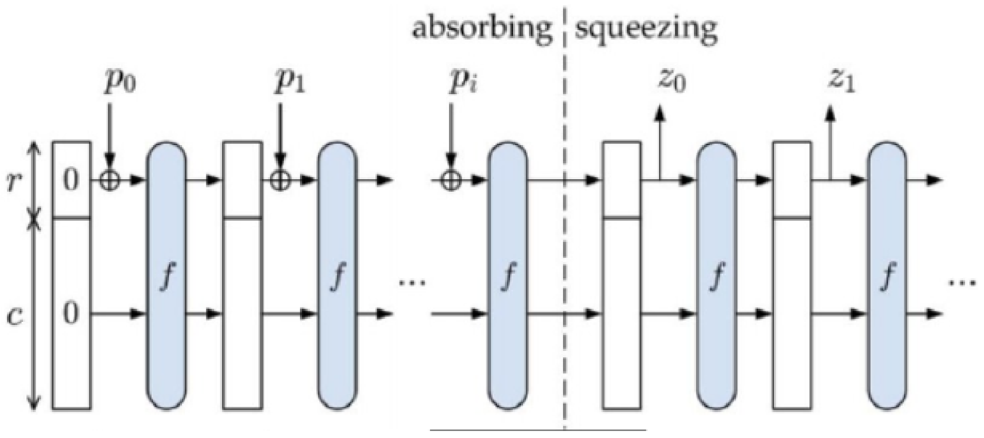


Figure 2.22: **Mechanism of a Sponge-Function took from** [4]

2. ρ is a rotation by an offset that depends on the word position, and π is a permutation

$$Rho(\rho) - Pi(\pi) : B[y, 2x + 3y] = ROT(A[x, y], r[x, y]) \qquad 0 \leq x, y \leq 4$$

3. χ as shown in the following equation consists of bitwise XOR, NOT, and AND operations.

$$Chi(\chi) : A[x, y] = B[x, y] \oplus ((NOT(B[x + 1, y]))AND(B[x + 3, y])) \qquad 0 \leq x, y \leq 4$$

4. Finally, ι is a constant round (RC) addition.

$$Iota(\iota) : A[0,0] = A[0,0] \oplus RC$$

When these five processes are completed, a round is completed.

The Keccak function has three different stages, initialization, absorption, and squeezing 2.22. In the absorption phase, the r-bit input message blocks are XORed with the first r-bit of the state and the resulting outputs are interrelated with the function f. When all message blocks are processed, the sponge's construction changes to the compression phase. In the compression phase, the first r-bit of the state is returned as an output block and is also inserted with the function f. The number of output blocks can be arbitrary and selected by the user.The Round Constant function RC[i] are given in Table 1and consists of 24 permutations values that assign 64 bit of data to Keccak function.

2.5 Asymmetric Cryptography

There is a second type of cryptography named asymmetric cryptography: in this kind of cryptography, **key1 $\neq$ key2** (visual representation in figure 2.23), if one is used for encryption, the other one must be used for decryption. Since there is not only one key, the keys are named according to the way in which they are stored: one is kept private (**private key**), while the other one is public (**public key**).

If a message is encrypted with a private key(Kpri) and sent to the receiver, it will use the corresponding public key(Kpub) to decrypt the message and

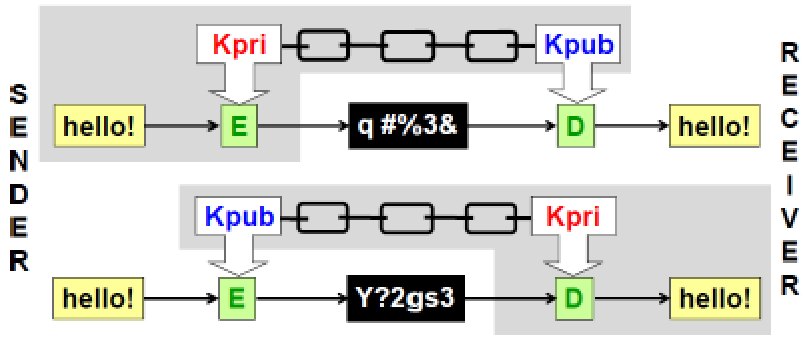


Figure 2.23: **Asymmetric Cryptography**

viceversa. One of the main advantages is that it eliminates the need to exchange secret keys, which can be a challenging process, especially when communicating with multiple parties. Additionally, asymmetric encryption is an important part in the creation of **digital signatures**, which can be used to verify the authenticity of data (**authenticity property**).

2.5.1 RSA

The RSA (Rivest-Shamir-Adleman) algorithm is a widely used asymmetric cryptographic algorithm for encryption and digital signatures. RSA functioning is based on three important step: Key Generation, encryption and decryption.

Key Generation

In this step the algorithm will generate a public key and a private key, based on the work provided by [19] we can summarize the creation of RSA keys as follows:

1. Generate two prime number, p and q.
2. $n = p \cdot q$. And p has to be $\neq q$, because if $p = q$ then $n = p_2$ so p can be obtained by simply pulling the square root of r.
3. Count $\varphi(n) = (p-1)(q-1)$
4. Choose public key, e, relatively prime to $\varphi(n)$.
5. Generate secret key d, $d \cdot e \equiv 1 \pmod{\varphi(n)}$.

So in the end, the RSA key generation algorithm assigns(e, n) as public key and d as private key.

Encryption

The RSA encryption algorithm uses the exponential function in modular n as in the following equation: $C \equiv P^e \pmod{n}$ to produce a ciphertext. Where C stands for Ciphertext and P stands for plaintext.

Decryption

The RSA decryption algorithm is an inverse of RSA encryption. Just like the encryption algorithm, the RSA decryption algorithm is a modular exponential function n by using the private key as in the following equation : $P \equiv C^d \pmod{n}$

RSA security property relies on the mathematical problem of factoring large composite numbers,that is" the decomposition of a positive integer into a product of interest which is still considered to be computationally hard to break " [56].

2.5.2 DSA

The Digital Signature Algorithm (DSA) stands as a pillar of modern cryptography, providing a robust framework for digital signatures (discussed in chapter 2.5.4). Developed by the National Institute of Standards and Technology (NIST) in the early 1990s, DSA offers a secure and efficient method for ensuring the integrity and authenticity of digital messages. DSA is an asymmetric cryptography algorithm where each entity (signer and verifier) possesses a pair of keys: a private key for signing and a public key for verification. The security of DSA is based on the computational difficulty of certain mathematical problems, particularly the discrete logarithm problem [44] in finite fields.

For a better understanding of the implementation of DSA we can refer to [19]. In DSA algorithm, there are 3 important steps : key generation, signing, and verifying

Key Generation

For the key generation we have to declare several parameters:

1. p, is a prime number(so a public parameter) with L (bit length), $512 \leq L \leq 1024$ and L must be multiples of 64.
2. q, is prime number factor of $p\check{}1$.
3. $g \equiv h^{\frac{p-1}{q}}$ the parameter g is a public one
4. x, is an integer less than q. x parameter is the private one.
5. $y \equiv g^x \pmod p$ is the public key.
6. m that is the message to be signed.

.

After all those initialization the key generation algorithm works as follow:

1. **Generating private key**: Randomly select a private key x such that $0 < x < q$.
2. **Calculating the Public Key**: Compute the public key $y$using the equation: $y = g^x modp$

Signing

Signers and verifiers must first agree to choose the same hash h function. To get its digital signature, the signer runs a signing algorithm using the hash h function and inputs a message m, private key, and public key

1. Generate number k randomly for each message, where $0 < k < q$
2. Count $r = (g^k modp)modq$
3. Count $s = (k^-1(SHA - l(m) + x * r)modq)$, where $SHA - l(m)$ is SHA hash function to m message.
4. The digital signature is (r, s)

Verifying

After the verifier receives the message and digital signature (m,r,s),runs a verifying algorithm to verify the digital signature.

1. Count $w = ((s)^-1modq)$

2. Count $tu1 = ((SHA - l(m) * w) mod q)$
3. Count $u2 = ((r * w) mod q)$
4. Count $v = (((g^u 1 * y^u 2) mod p) mod q)$
5. The digital signature is valid if $v = r$

2.5.3 ECDSA

ECDSA stands for Elliptic Curve Digital Signature Algorithm and is a Digital Signature Algorithm (DSA) which uses keys derived from elliptic curve cryptography (ECC). It is a particularly efficient equation based on public key cryptography . As suggested by [1] ECDSA provides a high degree of security with short key lengths as ECDSA uses less computation power. At the heart of ECDSA lies the concept of elliptic curves, which are mathematical structures defined by the equation

$$y^2 = x^3 + ax + b$$

over a field K, an elliptic curve has points that can be defined with coordinates (x, y) by giving values in a and b, we will call those points **point of origin**. From now on i will analyze the key Generation, the signature generation and the signature verification based on the work by [15]

Key Generation

The key generation process involves selecting a random private key d and computing the corresponding public key

$$Q = d * G$$

where G is an origin point on the elliptic curve.

Signature Generation

To sign a message m, an entity A does the following things:

1. Select a random or pseudorandom integer k where $1 \leq k \leq n - 1$ and n is the order of the curve.
2. Compute $kG = (x1, y1)$

3. Compute $r = x1modn$

4. Compute k^-1modn

5. Compute and Hash on the message computing for example SHA1(m) and convert the result into an integer, we will refer to this integer as e

6. Compute $s = k^-1(e+dr)modn$ The signature for the message m is (r,s).

Signature Verification

To verify Alice's signature (r,s) on m, Bob obtains an authentic copy of Alice's domain parameters and the public key Q. The verification process works as follow:

1. Compute SHA-1(m) and convert this bit string to an integer e

2. Compute $w = s^{-1}modn$

3. Compute $u_1 = ((ew)modn)$ and $u_2 = ((rw)modn)$

4. Compute $X = u_1G + u_2Q$

If X = 0, then reject the signature. Otherwise, convert the x-coordinate x1 of X to an integer i and compute $v = imodn$, if v=r then the signature is accepted.

"The security of ECDSA relies on the computational difficulty of solving the elliptic curve discrete logarithm problem, which is believed to be computationally hard to break."[47]

2.5.4 Digital Signature

A digital signature is a mathematical scheme for verifying the authenticity of digital messages or documents. A valid digital signature on a message gives a recipient confidence that the message came from a sender known to the recipient and for that reason it can provide the authenticity and non repudiation property and like handwritten signatures in the physical world, digital signatures serve as a seal of approval. For a better understanding of how a digital signature is performed we refer to the work by [59]. A digital signature scheme typically consists of three algorithms:

1. A key generation algorithm that selects a private key uniformly at random from a set of possible private keys.
2. A signing algorithm (such as DSA or RSA) that, given a message and a private key, produces a signature.
3. A signature verifying algorithm that, given the message, public key and signature, either accepts or rejects the message's claim to authenticity.

Usually a digital signature is not computed over the message that we want to sign but over the hash of the message and there are several reasons to sign such a hash (or message digest) instead of the whole document:

1. **Efficiency**: The signature will be much shorter and thus save time since hashing is generally much faster than signing.
2. **Compatibility**: messages are typically bit strings, but some signature schemes operate on other domains (such as, in the case of RSA, numbers modulo N). A hash function can be used to convert an arbitrary input into the proper format.
3. **Entirety**: Without the hash function, the text "to be signed" may have to be split in blocks small enough for the signature scheme to act on them directly. However, the receiver of the signed blocks is not able to recognize if all the blocks are present and in the appropriate order.

2.5.5 Algorithms security pillars

Is worth noticing that a lot of algorithms relies on security properties that ensure the safeness of the algorithm, in this section there will be a discussion about them, for a better comprehension on the general security level of a cryptographic algorithm.

- The **Integer Factorization Problem**, introduced in [36] said that altought The Fundamental Theorem of Arithmetic states that every positive integer can be expressed as a finite product of prime numbers and this factorization is unique except for the ordering of the factors, this existence proof gives no clue about how to efficiently find the factors of a large given integer. No polynomial time algorithm for solving this problem is known and by relying on this property we can build secure algorithms.

- The **Discrete Logarithm Problem**, introduced in [54] says that for large prime numbers p, computing discrete logarithms of elements of its multiplicative group (Z/pZ)is at present a very difficult problem. The security of certain cryptosystems is based on the difficulty of this computation.

- The **elliptic-curve discrete logarithm problem**: given a elliptic-curve cryptographic systems, e represents the selected random number that is the private key, G represents the generator point and P the derived public key, the elliptic curve discrete logarithm problem (ECDLP) is defined as the problem of finding an integer value e, such that the scalar multiplication of a primitive element G with e, produces another point P on the elliptic curve.

.

Chapter 3

Security Protocols

Security protocols play a pivotal role in safeguarding sensitive information and ensuring the integrity, confidentiality, and authenticity of digital communications in today's interconnected world. With the ever-growing prevalence of digital technologies and the increasing sophistication of cyber threats, the development and implementation of robust security protocols have become imperative for organizations, individuals, and societies at large.

This chapter aims to provide a comprehensive overview of the most used security protocols, examining their fundamental principles, underlying technologies, and practical applications. By exploring the various types of security protocols, their design considerations, and their strengths and limitations.

Throughout this chapter, we will delve into the key components of security protocols, including cryptographic algorithms, authentication mechanisms, and communication protocols. We will examine the role of encryption in ensuring confidentiality and privacy, the importance of digital signatures in verifying the authenticity of messages, and the significance of access control mechanisms in enforcing authorization policies.

Furthermore, we will explore the evolution of security protocols over time, tracing their development from early cryptographic techniques to modern standards and protocols such as TLS and SSH.

Finally, this chapter will discuss emerging trends and challenges in the field of security protocols, such as the proliferation of Internet of Things (IoT) devices and the rise of quantum computing. By addressing these issues, this chapter will give insights into the future direction of security protocols and the measures needed to ensure their continued effectiveness and resilience in the face of evolving threats and technological advancements.

3.1 Public Key Certificates

Before analyzing some of the most used security protocols is useful for a better understanding, to know what a Public Key certificate is, since they serves as an indispensable tools of the digital security. Public Key Certificates, often referred to simply as certificates, are an electronic document that provides a way to verify the authenticity of public keys and the identities associated with them, forming the cornerstone of many cryptographic protocols and systems. The certificate includes the public key and information about it, information about the identity of its owner (called the subject), and the digital signature of an entity that has verified the certificate's contents (called the issuer). If the device examining the certificate trusts the issuer and finds the signature to be a valid signature of that issuer, then it can use the included public key to communicate securely with the certificate's subject. In a typical public-key infrastructure (PKI) scheme, the certificate issuer is a certificate authority (CA), usually a company that charges customers a fee to issue certificates for them. The most common form of public key certificate is the **X.509 certificate**.

Some common fields that can be found in a Public Key Certificate are:

1. **Version Number**: It defines the X.509 version that concerns the certificate

2. **Serial Number**: Used to uniquely identify the certificate within a CA's systems.

3. **Subject**: the entity to which the certificate belongs. Usually it contains the subject name and the subject public key.

4. **Issuer**: The entity that verified the subject's information and signed the certificate.

5. **Key Usage**: The valid cryptographic uses of the certificate's public key. Common values include digital signature validation, key encipherment, and certificate signing.

6. **Public Key**: A public key belonging to the certificate subject.

7. **Signature Algorithm**: This contain a hashing algorithm and a digital signature algorithm. For example "sha256RSA" where sha256 is the hashing algorithm and RSA is the signature algorithm.
8. **Signature**:The body of the certificate is hashed and then the hash is signed with the issuer's private key.
9. **Validity Period**: It defines the period for which the certificate is valid.

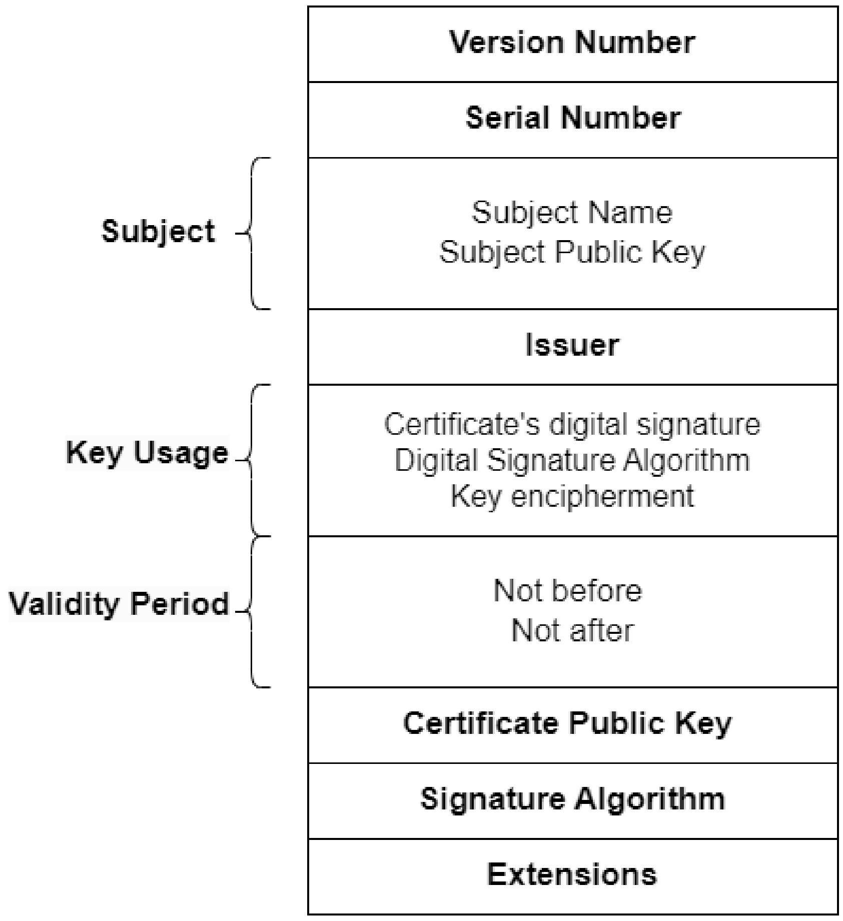


Figure 3.1: Structure of a X.509 certificate

3.1.1 Certicate Authorities

A certificate authority (CA) is the entity responsible for signing certificates. These certificates serve as a link between two parties, establishing trust in digital transactions. This trust is facilitated by a Certificate Authority (CA), which acts as a reliable third party in the process. A CA handles requests from individuals or organizations (known as subscribers) seeking certificates, verifying the provided information, and potentially issuing a certificate based on this verification. To effectively fulfill this role, a CA must possess one or more widely trusted root certificates or intermediate certificates, along with their corresponding private keys. Establishing broad trust usually involve including root certificates in popular software. Certificate authorities also bear the responsibility of keeping current revocation information for the certificates they issue, indicating their validity status. This information is obtained using protocols like the Online Certificate Status Protocol (OCSP) and/or Certificate Revocation Lists (CRL). The main differences in using those protocols are the following:

1. **Certificate Revocation List**: the certification authority creates a list of revoked certificates. The list is digitally signed by the CA, because otherwise someone could remove or add a certificate to the list to “block” someone. Since it is a list with all revoked certificates, this will permit to verify the validity since the certificate was issued. For example, if today a document is received and it was signed 3 months ago, we must check if the key was valid 3 months ago. If the certificate has been removed today, it does not affect the received document, because the check must be done against the time of the sign. The problem in using this protocol is that we must download the full list only to check one single certificate, but this is the only way to know the history of a certificate.

2. **Online Certificate Status Protocol**: this mechanism is preferred for real-time checks. In this case, we need to check if the certificate is valid at the time we want to verify it and the service needing the check do not care about previous states of the certificate. OCSP is a client-server protocol in which is possible to request to a specific service if the certificate is valid or not at the current time. The response is signed by the server, so that is not possible for someone to provide a fake response.

Some major software contain a list of certificate authorities that are trusted by default. This makes it easier for end-users to validate certificates, and easier for people or organizations that request certificates to know which certificate authorities can issue a certificate that will be broadly trusted. The policies and processes a provider uses to decide which certificate authorities their software should trust are called root programs. The most influential root programs are: Microsoft Root Program, Apple Root Program, Mozilla Root Program.

3.2 SSH

Secure Shell (SSH) is a protocol for secure remote login and other secure network services over an insecure network allowing remote login and command execution, as well as secure data communication between two devices. Since mechanisms like Remote Shell (a command-line computer program that can execute shell commands as another username, and on another computer across the network) are designed to access and operate remote computers, sending the authentication tokens (e.g. username and password) for the access requested by these computers that are in a public network in an unsecured way, poses a great risk of third parties obtaining the password and achieving the same level of access to the remote system as the remote user. Secure Shell mitigates this risk through the use of encryption mechanisms that are intended to hide the contents of the transmission from an observer, even if the observer has access to the entire data stream.

We refer to the original paper [64] e to the schema obtained from [37]for a better understanding. The protocol consists of three major components:

1. **The Transport Layer Protocol**, this layer handles initial key exchange as well as server authentication, and sets up encryption, compression, and integrity verification. The transport layer also arranges for key re-exchange, usually after 1 GB of data has been transferred or after one hour has passed, whichever occurs first.

2. **The User Authentication Protocol** authenticates the client-side user to the server. Authentication is client-driven, the server merely responds to the client's authentication requests. Widely used user-authentication methods includes: password, public-key-based authentication (usually supporting at least DSA, ECDSA or RSA keypairs), one-time password authentication or by providing external mechanism for the authentication.

3. **The Connection Protocol** defines the concept of channels, channel

requests, and global requests, which define the SSH services provided. A single SSH connection can be multiplexed into multiple logical channels simultaneously, each transferring data bidirectionally. Channel requests are used to relay out-of-band channel-specific data, such as the changed size of a terminal window, or the exit code of a server-side process.

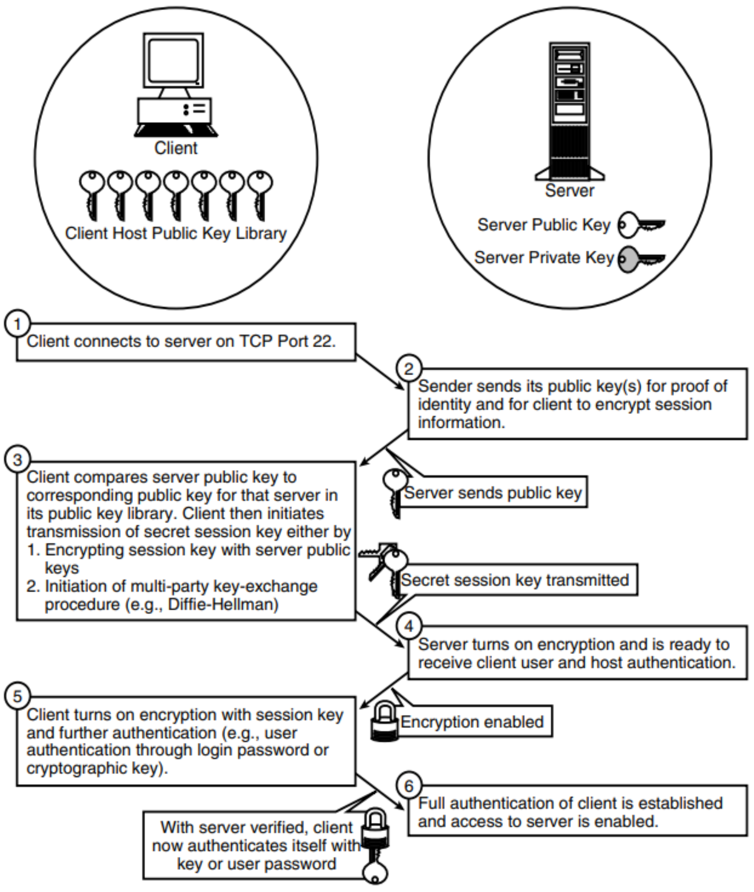

Client
Client Host Public Key Library
Server
Server Public Key
Server Private Key
1 Client connects to server on TCP Port 22.
2 Sender sends its public key(s) for proof of identity and for client to encrypt session information.
3 Client compares server public key to corresponding public key for that server in its public key library. Client then initiates transmission of secret session key either by 1. Encrypting session key with server public keys 2. Initiation of multi-party key-exchange procedure (e.g., Diffie-Hellman)
Server sends public key
Secret session key transmitted
4 Server turns on encryption and is ready to receive client user and host authentication.
5 Client turns on encryption with session key and further authentication (e.g., user authentication through login password or cryptographic key).
Encryption enabled
6 Full authentication of client is established and access to server is enabled.
With server verified, client now authenticates itself with key or user password

Figure 3.2: **SSH handshake schema from** [37]

In figure 3.2 we have a conceptual schema on how SSH authentication and asymmetric encryption works, divided in many steps as follows:

1. The client host (left) connects to the server host (right), conventionally on TCP port 22. The server and client exchange the protocol versions they support and, if compatible, the connection continues (at this stage is still unencrypted).

2. The server sends its authentication information and session parameters to the client. This includes the server host's public key component of its own public/private key pair, as well as a list of the encryption, compression, and authentication modes that the server supports. The Public key algorithms supported by most SSH implementations include RSA and DSA.

3. The client host checks the server host's public key against the client host's library of public keys. If this is the first time that this particular client and this particular server host have connected over SSH, the user is asked to verify the addition of new a server host public key to the client's public key library. Any future connections to that particular server host will now be verified against that public key reported from their first contact. Once the identity of the server is verified, a secret session key is generated.

4. Once the secret session key is in the possession of both parties, encryption and integrity checking are turned on. Session keys are typically stored only in memory and are not written to storage for security purposes

5. The client host and the client host user ("the user") can now be authenticated to the server host without fear of the authentication and access transmission being intercepted or corrupted in transit. Methods by which the users authenticate themselves to the server include plaintext user login passwords, a user public key certificate or other preferred methods. Once the user is authenticated, appropriate access to the server and its services are granted to the user. The two machines are now connected through a secure, encrypted connection which can be used as a secure tunnel.

3.3 IPsec

Internet Protocol Security (IPsec) is a secure network protocol suite that authenticates and encrypts packets of data to provide secure encrypted communication between two computers over an Internet Protocol network. It supports network-level peer authentication, data origin authentication, data integrity and data confidentiality.

IPsec is an open standard as a part of the IPv4 (the fourth version of the Internet Protocol, the ne that describes how the basic packets delivery is done, essentially establishing the internet) suite and uses the following protocols to perform various functions:

1. **Authentication Header (AH)**: provides data integrity and data origin authentication for IP packets while also providing protection against replay attacks (A replay attack is a form of network attack in which valid data transmission is maliciously or fraudulently repeated or delayed).

2. **Encapsulating Security Payload (ESP)** provides confidentiality, data integrity, data origin authentication, and limited traffic-flow confidentiality.

The IPsec protocols AH and ESP can be implemented in a host-to-host transport mode, as well as in a network tunneling mode.

3.3.1 Transport Mode

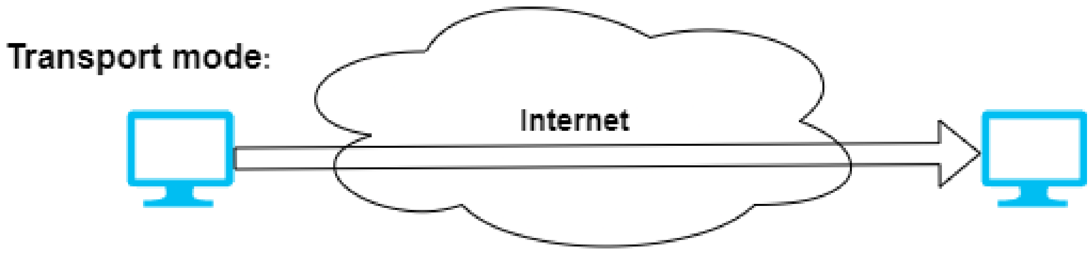


Figure 3.3: **Transport mode**

It is used for end-to-end security. In transport mode, only the payload of the IP packet is usually encrypted or authenticated. The routing process is intact, since the IP header is neither modified nor encrypted; however, when the authentication header is used, the IP addresses cannot be modified , as this always invalidates the hash value carried inside the package.

3.3.2 Tunnel Mode

In tunnel mode, the entire IP packet is encrypted and authenticated. It is then encapsulated into a new IP packet with a new IP header. Tunnel mode

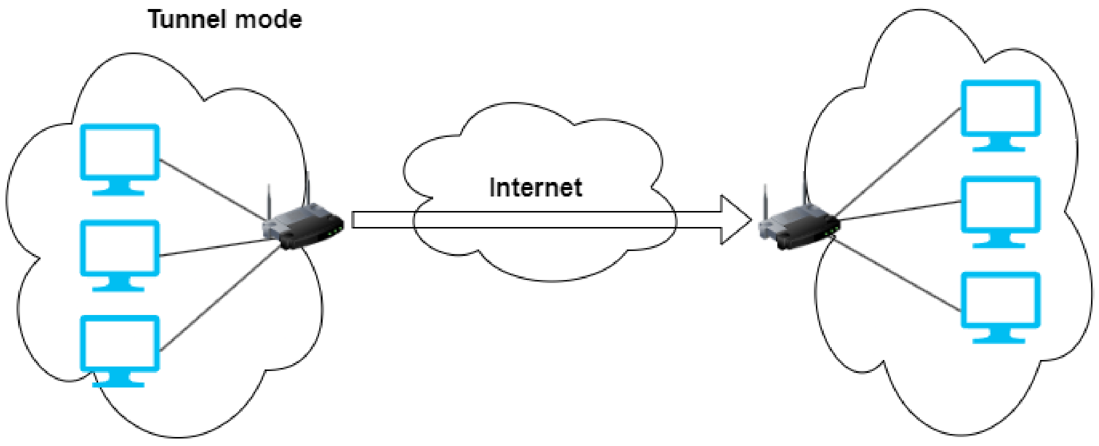


Figure 3.4: **Tunnel mode**

is used to create virtual private networks (a mechanism for creating a secure connection between a computing device and a computer network using an insecure communication)

3.3.3 Authentication Header

The following picture shows how an AH packet is constructed and interpreted:

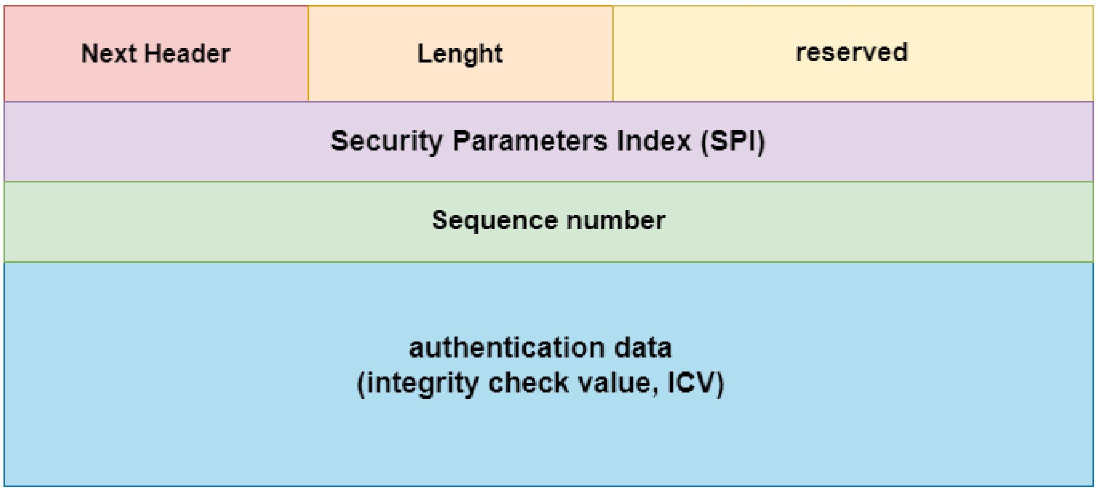


Figure 3.5: **IPsec AH header format**

The format of the authenticaton header includes:

1. **Next header** field because this is a pseudo-protocol, so in the IP header there will be written that it is transporting AH, but then inside the AH there is the real transporting packet field

2. **Lenght** parameter of 1 byte to describe the length of the packet

3. **Reserved** bytes for future uses

4. **Security Parameters Index (SPI)**: 32 bits for referring in a quick and easy way to all the parameters that are needed to verify in the packet

5. **Sequence number** to avoid replay attacks

6. **Integrity Check Value (ICV)**: variable number of 4 bytes words to provide authentication data

When an IPsec packet is received it is protected with AH. It starts with the extraction of the AH and from it the ICV is extracted, which is the received authentication value (the digest computed by the sender). Then, on the received packet the normalization is performed, which means to put the packet in the same condition as it was at the sender in order to compute the same kind of hash. Once the normalized IP packet is available, it is needed to compute the authentication value (ICV). For that, the Security Parameter Index (SPI) is being used inside the Database of the Security Association (SAD). It is a pointer that indicates algorithm and parameters to be used. These parameters can be used to compute the authentication value and then it is checked if the two values (the one computed and the one received from the sender) are the same. If the two values are equal, then the sender is authentic, and the packet is integral. If the two values are not equal, there could be a fake sender and/or manipulated packet. An authentic sender is specified in the previous picture, but there is no place where the sender is authenticated. So, who is the sender authenticated here? The answer is in the fact that is being used a specific entry in the SAD. That entry negotiated with a specific node. For this reason, authentication is implicit in the process. The real authentication comes into play when we create the Security Association: this is the point in which the sender must prove its identity. Then the SA brings on that kind of authentication thanks to the usage of the correct algorithm/parameters.

3.3.4 Encapsulating Security Payload

If confidentiality is wanted, Encapsulating Security Payload (ESP) is needed. The base mechanism works on DES-CBC ([43]), but other mechanisms are also possible. The advantage over AH is that the packet dimension is reduced. In the following picture we can see how an ESP packed is made:

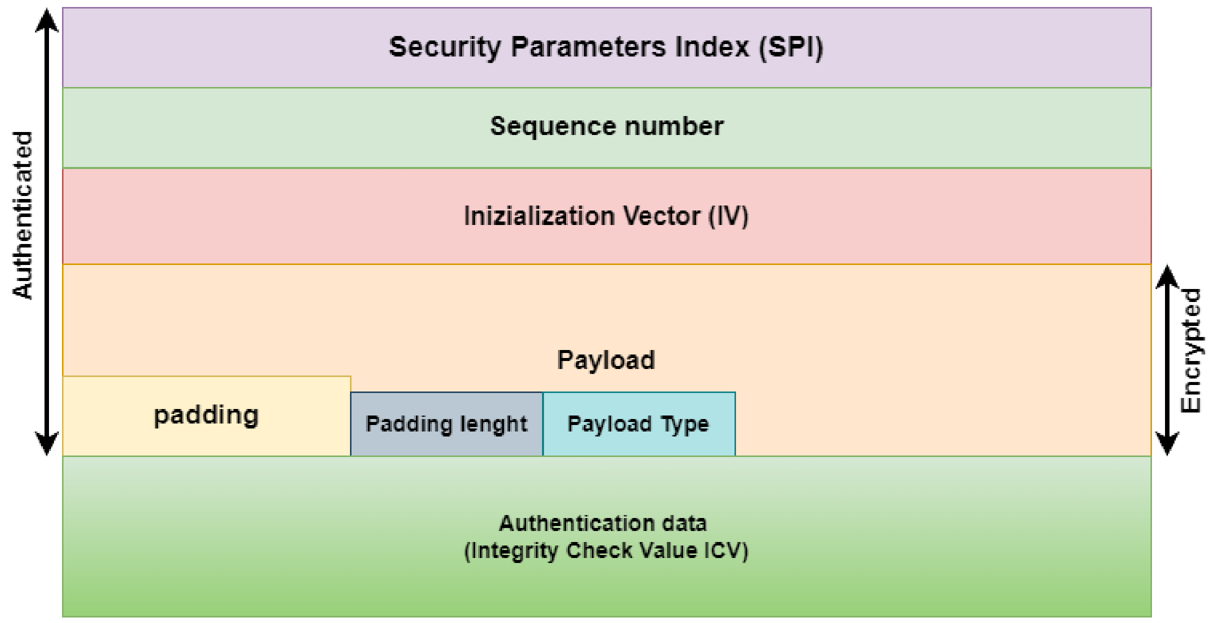


Figure 3.6: **IPsec ESP header format**

Only the Security Parameterd Index and the sequence number are in clear informations, all the rest are encrypted data, let's assume we are using DES in CBC mode. Since it is being used DES-CBC an Initialization Vector (IV) is required. Then there is the payload itself. The part from the SPI to the padding (to reach a multiple of 64 bits) is the authenticated one. There must be also 1 byte to declare Padding length and 1 byte for the Payload Type, which is the layer 4 protocol that we are transporting.

3.4 TLS

Transport Layer Security (TLS) is a cryptographic protocol designed to provide communications security over an untrusted computer network. The protocol is widely used in applications such as email and instant messaging, but its use in securing HTTPS remains the most publicly visible. The TLS protocol aims primarily to provide confidentiality, integrity, and authenticity through the use of cryptography and the use of certificates, between two or more communicating computer applications. TLS is a proposed Internet Engineering Task Force (IETF) standard, first defined in 1999, and the current version is TLS 1.3, defined in August 2018 [49]. TLS builds on the now-deprecated SSL (Secure Sockets Layer) [3].

Client-server applications use the TLS protocol to communicate in a secure way across an untrusted network, since applications can communicate either with or without TLS (or SSL), it is necessary for the client to request that the server sets up a TLS connection and the first step for doing it is by using some specific **port number** (a port number is a number assigned to uniquely identify a connection endpoint and to d a specific service) , where port 80 is used for unencrypted traffic and port 433 is the common port used for the TLS protocol. Once the client and server have agreed to use TLS, they negotiate a connection by using the **handshake protocol**. During this handshake, the client and server agree on the parameters that are going to be used to establish the connection's security:

1. The handshake begins when a client connects to a TLS-enabled server requesting a secure connection and the client presents a list of supported cipher suites (a set of algorithms that can secure the network) and from this set the server choose the best supported ciphersuite.

2. The server also provides identification using a **Digital Certificate** and the client will validate the certificate.

3. If the client confirms the validity of the certificate, it will generate the session keys for securing the connection by either: **encrypting a random number** (PreMasterSecret) with the server's public key (previously obtained from the public certificate) and sends the result to the server. Then both client and server will use the random number to generate a unique session key for the future encryption and decryption of the session. Or using the **Diffie-Hellman key exchange** to securely generate a random and unique session key for encryption and decryption that also has the additional property of **forward secrecy**: if the server's private key is disclosed by an attacker, then it will not be able to use it to decrypt the current session's traffic but only the past one.

This concludes the handshake and begins the secured connection, which is encrypted and decrypted with the session key until the connection closes. If any one of the above steps fails, then the TLS handshake fails and the connection is not created.

When a connection uses the TLS protocol it gains:

1. **confidentiality** thanks to the symmetric key encryption algorithm used to encrypt the data, moreover those keys are generated uniquely for each connection and are based on a shared secret that was negotiated at the start of the session.

2. **authentication** : using public-key cryptography. Authentication is mandatory only for the server and optional for the client.

3. **integrity**: each message transmitted includes a message authentication code to prevent undetected loss or alteration of the data during transmission.

3.4.1 TLS record

The TLS protocol exchanges **records**, which encapsulate the data to be exchanged in a specific format. Each record can be compressed, padded, appended with a message authentication code (MAC), or encrypted, all depending on the state of the connection.

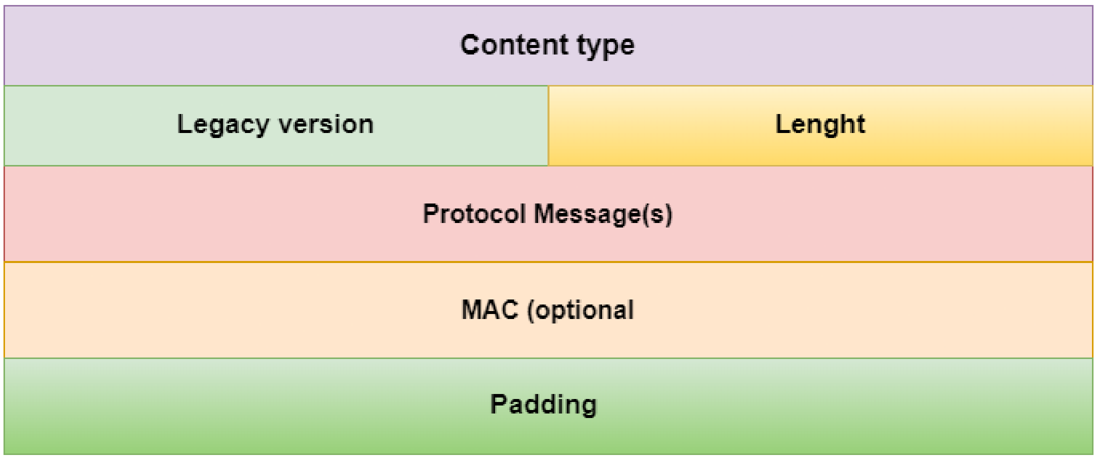


Figure 3.7: **General TLS Record Format**

In figure 3.7 we can see the main fields that a TLS record has:

1. **Content type** serves as an identifier of the protocol to which this record refers, it can assume multiple values, but the most common are: 20 that means the ChangeCipherSpec protocol, 21 for the alert protocol and 22 for the handshake protocol.
2. **Legacy version**: identifies the major and minor version of TLS prior to TLS 1.3 for the contained message.
3. **Length**: The length of the protocol message(s), MAC and padding fields combined, that must not exceed 2^{14} bytes.
4. **Protocol message**: One or more messages identified by the Protocol field.
5. **MAC and padding**: A message authentication code computed over the "protocol message(s)" field, with additional key material included.

3.4.2 TLS Alert Protocol

This message is based on the standard TLS record explained before but with a difference in the structure as showed in figure 3.8:

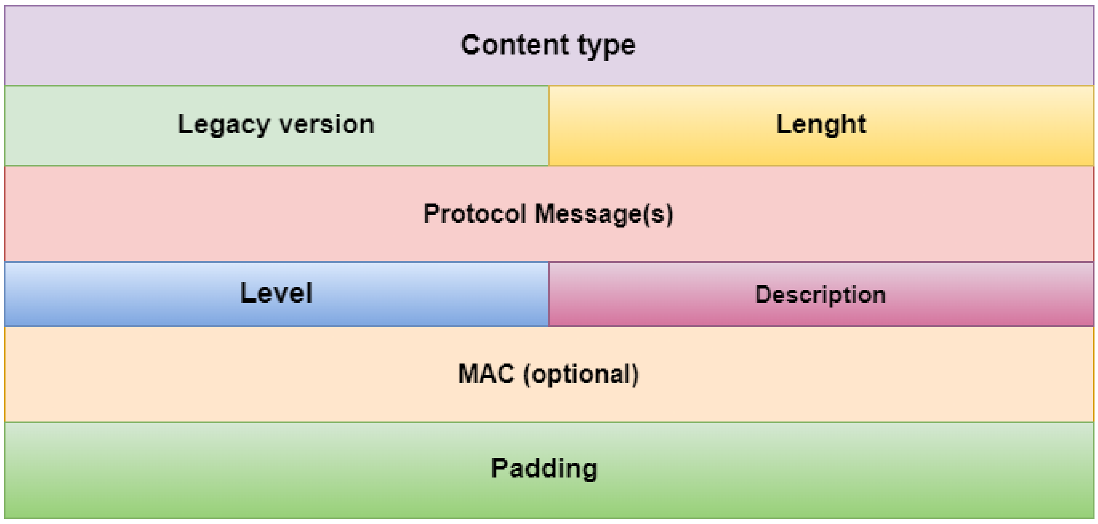


Figure 3.8: **TLS record format for alert protocol**

This message can be sent at any time during the handshake and up to the closure of the session and it's used to signal an error during the handshake procedure. After this message the tls session will be closed and this record can give to the user a reason for the closure. The record has 2 additional fields called Level and Description.

1. Level is used to identify the level of the alert, and can assume two values: 1 or 2, corresponding to warning (the connection may be unstable) and fatal (an unrecoverable error has occurred)

2. Description identifies which type of alert is being sent. It can assume multiple values but some or the most common are:

   - **10**, unexpected message.
   - **21**, decryption failed.
   - **40**, handshake failure.
   - **42**, Bad certificate.
   - **48**, unknown Certificate Authority.
   - **71**, insufficient security.
   - **116**, certificate required.

The use of Alert records is optional, however if it is missing before the session closure, the session may be resumed automatically (with its handshakes).

3.4.3 TLS handshake

A typical messages exchange in the TLS handshake protocol works as illustrated in figure 3.9:

A client sends a ClientHello message specifying the highest TLS protocol version it supports, a random number, a list of suggested cipher suites and suggested compression methods. The server responds with a ServerHello message, containing the chosen protocol version, a random number, cipher suite and compression method from the list previously obtained by the client. The chosen protocol version should be the one that grants highest that security level obtainable by the client and server. The server sends its Certificate message (depending on the selected cipher suite, this may be omitted by the server). The server sends a ServerHelloDone message, indicating it is done with handshake negotiation. The client responds with a ClientKeyExchange message, which may contain a **PreMasterSecret**, public key, or nothing. This PreMasterSecret is encrypted using the public key of the server, the client has access to the server's public key, by looking

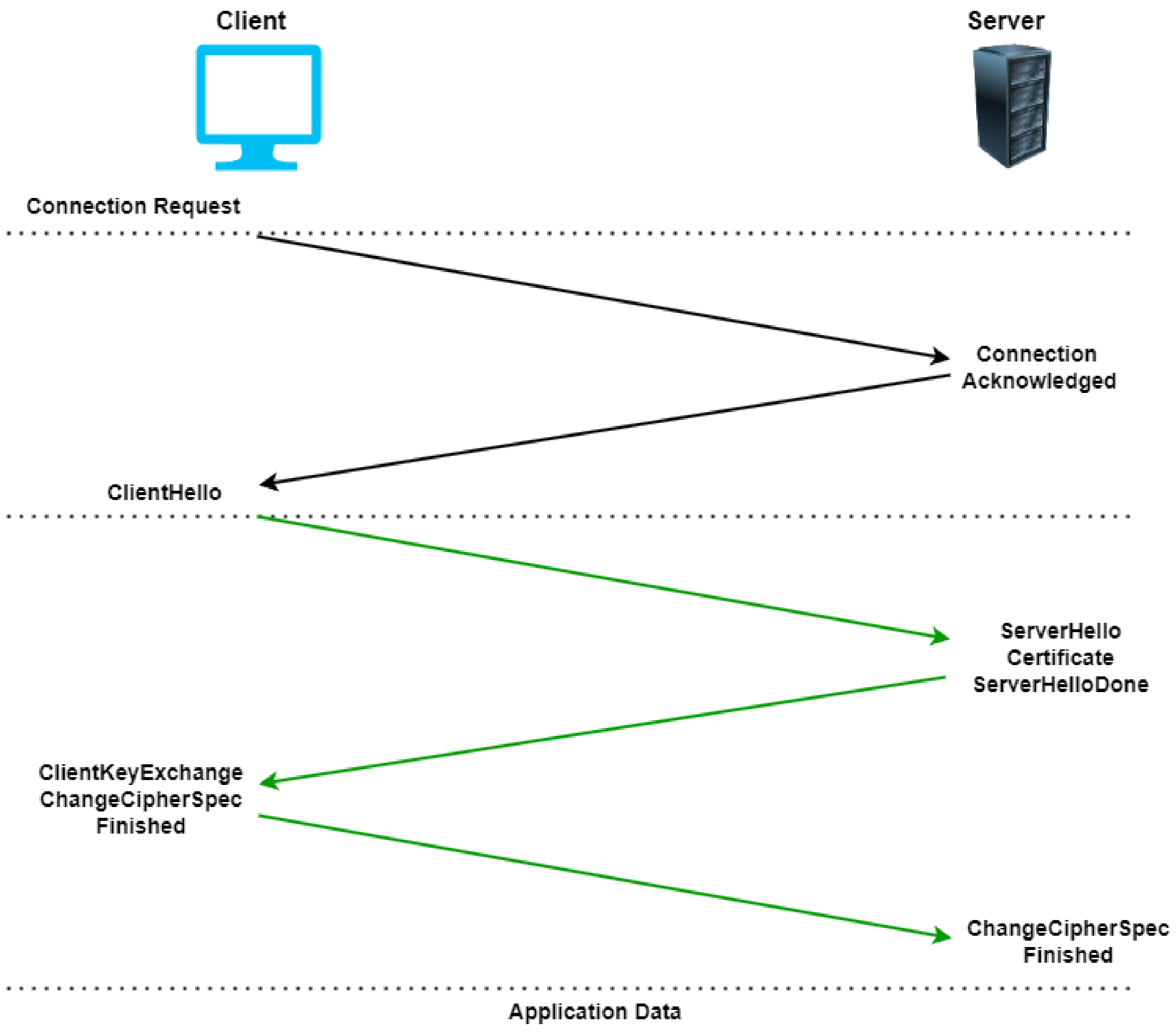


Figure 3.9: **Basic TLS Handshake**

at its certificate. The client and server then use the random numbers and PreMasterSecret to generate a common secret, called the **master secret**. All other key data (session keys such as IV, symmetric encryption key, MAC key) for this connection are derived from this master secret and the client and server generated random values. The client now sends a ChangeCipherSpec record, essentially telling the server that every message exchanged from now on will be authenticated and encrypted. The client sends an authenticated and encrypted Finished message, containing a hash and MAC over the previous handshake messages. The server will attempt to decrypt the client's Finished message and verify the hash and MAC. If the decryption or verification fails, the handshake is considered to have failed and the connection should be terminated. Finally, the server sends a ChangeCipherSpec and its authenticated and encrypted Finished message. The client performs the same decryption and verification procedure as the server did in the previous step. Application phase: at this point, the "handshake" is completed and the application protocol is enabled, with content type of 23. Application messages exchanged between client and server will also be authenticated and optionally encrypted exactly like in their Finished message.

Sometimes also the client authentication is requested by the server so the protocol will have some differences, figure 3.10 shows an message flow of a mutual-authentication TLS handshake.

A client sends a **ClientHello** message specifying the highest TLS protocol version it supports, a random number, a list of suggested cipher suites and suggested compression methods.

The server responds with a **ServerHello** message, containing the chosen protocol version, a random number, cipher suite and compression method from the list previously obtained by the client.

The server sends a**CertificateRequest** message, to request a certificate from the client and then a **ServerHelloDone** message, indicating it is done with handshake negotiation. The client responds with a **Certificate message**, which contains the client's certificate, but not its private key.

The client sends a **ClientKeyExchange** message, which may contain a **PreMasterSecret**, public key, or nothing (depending on the selected ciphersuite). This PreMasterSecret is encrypted using the public key of the server, the clint has access to the server's public key, by looking at its certificate.

The client sends a **CertificateVerify** message, which is a signature over the previous handshake messages using the client's certificate's private key. This signature can be verified by using the client's certificate's public key. This lets the server know that the client has access to the private key of the certificate and thus owns the certificate. The client and server then use the random numbers and PreMasterSecret to compute a common secret, called the **master secret**. All other key data (from now on called "session keys") for this connection is derived from this master secret (and the client- and server-generated random values), which is passed through a carefully designed pseudorandom function.

The client now sends a **ChangeCipherSpec** record, essentially telling

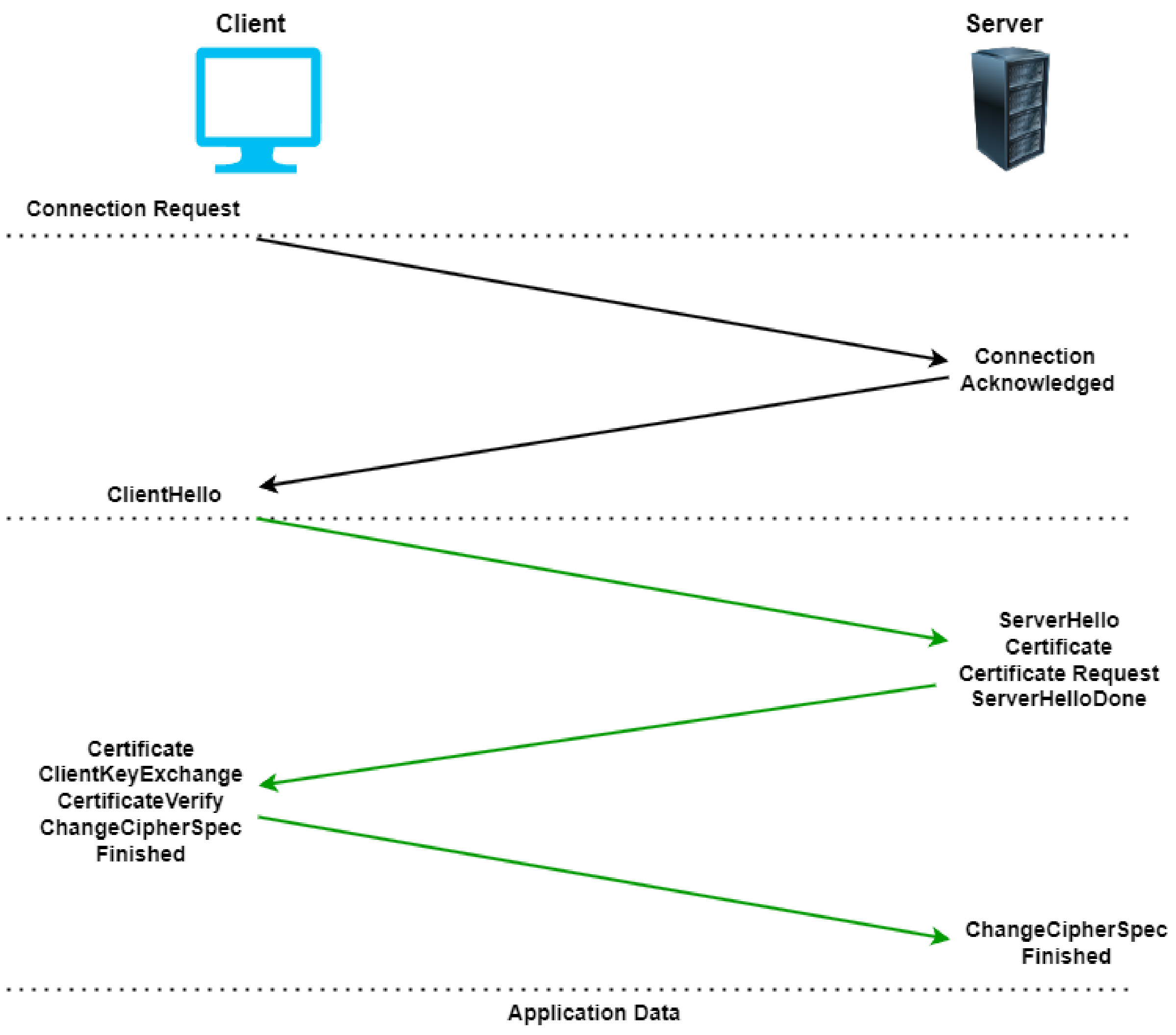


Figure 3.10: **Basic TLS Handshake with mutual Authentication**

the server, "Everything I tell you from now on will be authenticated and encrypted". The client sends an authenticated and encrypted **Finished** message, containing a hash and MAC over the previous handshake messages. The server will attempt to decrypt the client's Finished message and verify the hash and MAC. If the decryption or verification fails, the handshake is considered to have failed and the connection should be terminated. Finally, the server sends a **ChangeCipherSpec** and its authenticated and encrypted Finished message. The client performs the same decryption and verification procedure as the server did in the previous step.

3.4.4 TLS 1.3

TLS 1.3 is the current version of the protocol, defined in August 2018 [49], is condensed to only one round trip compared to the two round trips required in previous version of TLS/SSL. TLS 1.3 has rapidly been adopted giving a significant contribution to the Internet [27]. In fact, it deprecates vulnerable cryptographic primitives and substantially reduces the time required to perform the handshake compared to the TLS 1.2 handshake. Compared to TLS 1.2, TLS 1.3 guarantees perfect forward secrecy by removing static RSA key exchanges. It also reduces the number of round-trips of the TLS handshake from two to one, aiming to improve the performance of the initial setup. The following figure shows the sequence of messages for the full TLS handshake [12].

The handshake is mainly composed of three parts:

1. Key Exchange:The client sends a ClientHello message to server. The server processes the ClientHello message and determines the appropriate cryptographic parameters for the connection. It then responds with its own ServerHello message, which indicates the negotiated connection parameters. For TLS 1.3, the ServerHello message determines the key and cipher options only. Other handshake parameters may be determined later.

2. Server Parameters:The server sends two messages to establish server parameters: **EncryptedExtensions**: This message contains responses to ClientHello extensions that are not required to determine the cryptographic parameters, other than those that are specific to individual certificates. **CertificateRequest** (optional): If certificate-based client authentication is desired, then the server sends this message, which contains the desired parameters for that certificate. This message is omitted if client authentication is not desired.

3. **Authentication**: The server sends these authentication messages:

   (a) **Certificate** (optional): This message contains the authentication certificate and any other supporting certificates in the certificate chain. This message is omitted if the server is not authenticating with a certificate. **CertificateVerify** (optional): This message contains a signature over the entire handshake using the private key corresponding to the public key in the Certificate message. This

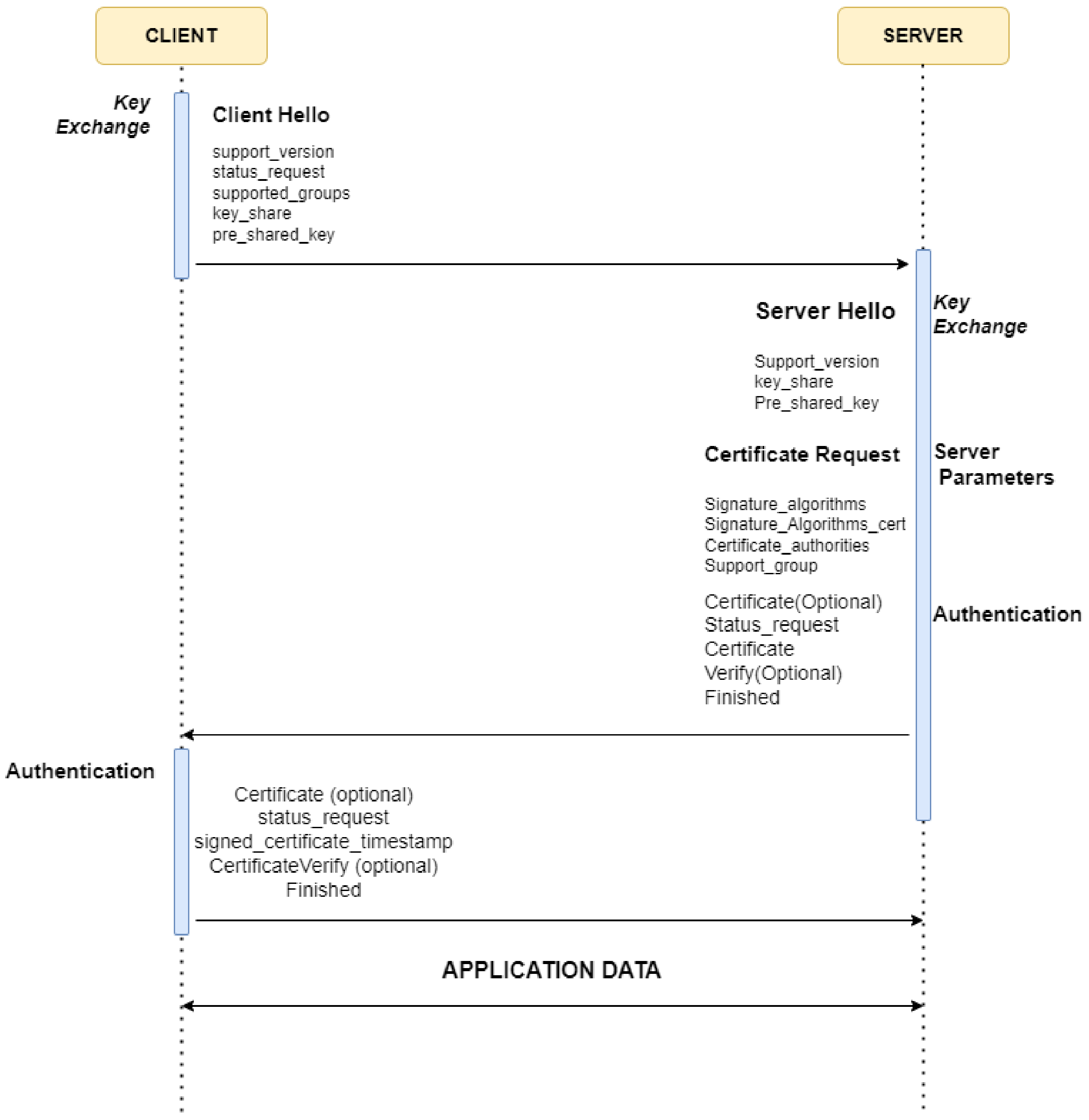


Figure 3.11: **TLS 1.3 handshake**

message is omitted if the server is not authenticating with a certificate. **Finished**: a MAC over the entire handshake.

The client responds with its own Certificate, CertificateVerify, and Finished messages. The Certificate message is omitted if the server did not send a CertificateRequest message. The CertificateVerify message is omitted if the client is not authenticating with a certificate.

Chapter 4

Post-Quantum Cryptography

In recent years, the research in post-quantum cryptography has been going through a significant improvement driven by the imminent threat of quantum computers. Traditional cryptographic schemes, which form the base of secure communication and data protection, rely on mathematical problems that are believed to be hard for classical computers to solve efficiently, such as the previously described (see Section 2.5.5) integer factorization, discrete algorithm, and elliptic-curve discrete logarithm problems. However, the advent of quantum computing threatens to undermine the security of these schemes by potentially rendering these problems solvable by a quantum computer running **Shor's Algorithm** [52]. The Shor's algorithm is a quantum algorithm for finding the prime factors of an integer, it can break the security pillars of an asymmetric algorithm such as the logarithm or the integer factorization problem, and it was developed in 1994 by Peter Shor. On the other hand, there is another algorithm called **the Grover's algorithm** that can break the security provided by symmetric algorithms since it can lead to a quadratic advantage for many interesting computational tasks such as unstructured search problems [28]. The Grover's algorithm is more difficult to apply compared to the Shor one since it needs more qubit to work. This leads to a different threat for asymmetric algorithms and the symmetric one: most of the current symmetric cryptographic algorithms are considered to be relatively secure against attacks by quantum computers by using a key of 256 bits or more[6]; on the other hand, the threat to asymmetric cryptography is tangible. While as of 2024, quantum computers lack the processing power to break widely used cryptographic algorithms [20], cryptographers are designing new algorithms to be prepared for **Q-Day**, the day when current algorithms will be vulnerable to quantum computing attacks [50]. The need for cryptographic solutions that can resist against attacks from quantum computers has led to the emergence of a new research area known as post-quantum cryptography. Post-quantum cryptography aims to develop cryptographic algorithms and protocols that remain secure even in the presence of powerful quantum adversaries.

This chapter provides an overview of post-quantum cryptography, covering its historical context and fundamental concepts. Furthermore, we delve into some of the families of post-quantum cryptographic algorithms, including lattice-based cryptography and hash-based cryptography.

Overall, this chapter is an introduction to post-quantum cryptography, providing a comprehensive overview of its significance, principles, and ongoing developments in the pursuit of secure communication in the quantum age.

A **quantum computer** is a computing device that uses the principles of quantum mechanics to perform certain types of computations much more efficiently than classical computers. The main characteristics of the qubits are: **Superposition**: qubits can exist in a superposition of both 0 and 1 states simultaneously. This allows quantum computers to process a large number of possibilities at the same time. **Quantum Parallelism**: quantum computers can exploit the parallelism provided by superposition to process a large number of possibilities simultaneously. This can lead to exponential speedup for certain types of computations.

As a result of the fast computation a quantum computer can achieve, widely used cryptographic algorithms such as RSA, which currently secure our digital transactions and communications, may become vulnerable to attacks from quantum adversaries in the next future. Their work has gained attention from academics and industry such as the European Telecommunications Standards Institute (ETSI) [18] and NIST (National Institute of Standards and Technology) [40] that is in charge of the standardization of the Post-Quantum algorithms.

NIST in the post-quantum standardization process started in 2017 [9] chose four valid post-quantum algorithms: **CRYSTALS-Kyber** for the Key encryption mechanism (based on cryptography on lattices) and **Dilithium, Falcon and SPHINCS+** to perform digital signature (the first two based on cryptography on lattices and the last based on hash functions). They are starting to define them naming it, respectively, as ML-KEM in FIPS 203, ML-DSA in FIPS 204, SLH-DSA in FIPS 205 and FN-DSA in FIPS 206 [10].

The NIST standardization body, in the Special Publication 800-131A [41], defined three security levels (level 1, level 3, Level 5) that correspond to different levels of security strength for cryptographic algorithms. Level 1 provides a basic level of security, similar to the level of security provided by AES-128, used for applications where the primary goal is to protect against casual threats. Level 3 provides a moderate level of security suitable for protecting sensitive information, similar to the level of security provided by AES-192. Level 5 provides a high level of security, similar to the one provided by AES-256, and is suitable for protecting highly sensitive information and systems [25].

4.1 Lattice-based cryptography

Lattice-based cryptography is the generic term for constructions of cryptographic primitives that involve lattices, either in the construction itself or in the security proof. Lattice-based constructions are one of the building blocks of post-quantum cryptography. Unlike public-key schemes based on: the integer factorization, the discrete logarithm and the elliptic curve problem (such as the RSA, Diffie-Hellman or elliptic-curve cryptosystems) which could be defeated using Shor's algorithm on a quantum computer, some lattice-based constructions appear to be resistant to attack by both classical and quantum computers. The NIST standardization process in the Post-Quantum field has identified two cryptographic schemes to be considered as "primary" solutions: regarding Key Encapsulation Mechanisms (KEMs), CRYSTALS-Kyber has been selected, while in the context of digital signatures, the choice has fallen on CRYSTALS-Dilithium.

The most important lattice-based computational problem is the **shortest vector problem** that can be visualized using figure 4.1: given a basis of a vector space V, and a norm N , find the shortest non-zero vector in V (the red arrow in figure 4.1). This problem is thought to be hard to solve efficiently, even with approximation factors, and even with a quantum computer.

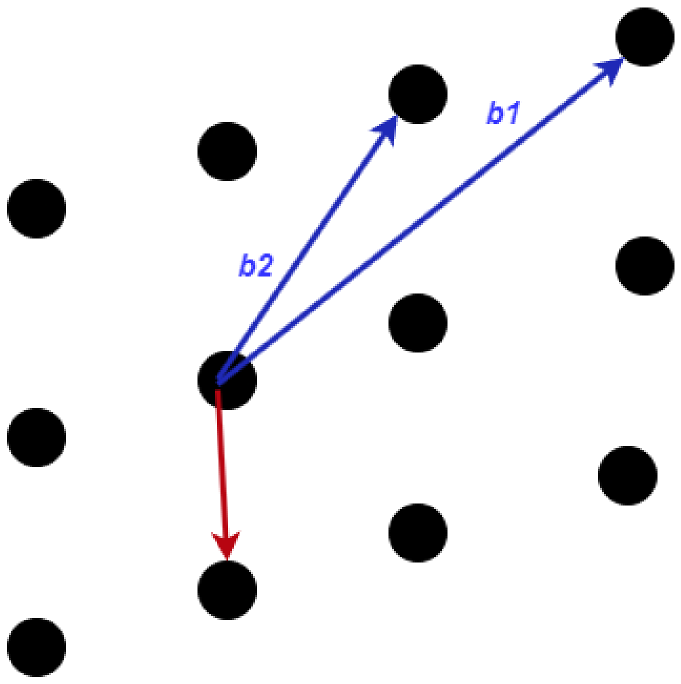


Figure 4.1: Illustration of the shortest vector problem

4.2 Hash-based cryptography

Hash-based cryptography is the generic term for constructions of cryptographic primitives based on the security of hash functions.

Hash-based cryptography is used to construct digital signatures, combining a **one-time signature scheme** with a **Merkle tree structure**.

A **one-time signature schema** is a digital signature scheme that can be used to sign one message per key pair, with no assurance of security if the key pair is reused to sign again. The one-time digital signature schemes have the advantage that **signature generation and verification are very efficient but the key must be recomputed any time**.

A **Merkle tree** structure is a tree in which every leaf node is labelled with the cryptographic hash of a data block, and every node that is not a leaf is labelled with the cryptographic hash of the labels of its child nodes as showed in: **??**.

Since a one-time signature scheme key can only sign a single message securely, it is practical to combine many such keys within a single, larger

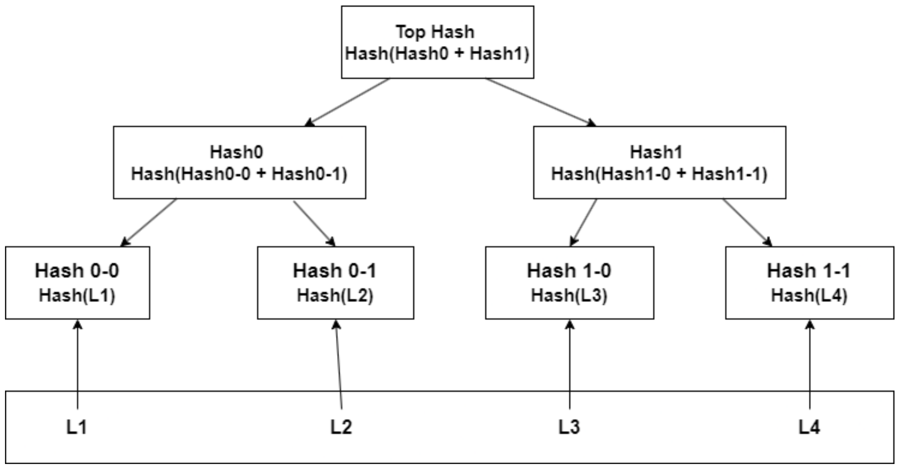


Figure 4.2: Merkle tree on the L1,L2,L3,L4 blocks

structure. In this hierarchical data structure, a hash function and concatenation are used repeatedly to compute tree nodes (Hash L1...HashL4 in figure **??** .

The central idea of hash-based signature schemes is to combine a larger number of one-time key pairs into a Merkle tree to obtain a practical way of signing more than once.

One public and one private key are constructed from the numerous public and private keys of the underlying one-time scheme. The global public key is the single node at the very top of the Merkle tree (the top node in figure **??**.

Its value is an output of the selected hash function, so a typical public key size is 32 bytes. The validity of this global public key is related to the validity of a given one-time public key using a sequence of tree nodes. This sequence is called the **authentication path**. It is stored as part of the signature, and allows a verifier to reconstruct the node path between those two public keys.

The global private key is generally handled using a pseudo-random number generator. Then it is sufficient to store a seed value. One-time secret keys are derived successively from the seed value using the generator.

Hash-based signature schemes can only sign a fixed number of messages securely, a maximum of 2^h where h is the height of the Merkle tree. For the reasons explained before, hash-based signature schemes can sign only a limited number of messages securely. The US National Institute of Standards and Technology (NIST), specified that algorithms in its post-quantum cryptography competition must support a minimum of 2^{64} signatures safely [38].

4.2.1 CRYSTALS-Kyber

Kyber is a post-quantum algorithm based on lattices. It is used to establish a shared secret between two communicating parties making an attacker unable to decrypt it. Its main use case is to establish keys of symmetric-key systems in higher-level protocols like TLS. Kyber has three security levels and the trade between key size and security levels are better shown table [23] using RSA as a pre-quantum comparison. The system is based on the module learning with errors (LWE) problem. Different versions of the Kyber algorithm have been defined based on the security levels needed: Kyber512 (NIST security level 1), Kyber768 (NIST security level 3), and Kyber1024 (NIST security level 5). Its keys are bigger than those of pre-quantum schemes, but small enough to be used in real-world systems. This makes Kyber an interesting candidate for many PQC applications. It won the NIST competition for the first post-quantum cryptography standard [55]. NIST calls its draft standard Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) [39].

Version	**Security Level**	**Private Key Size**	**Public Key Size**	**Ciphertext Size**
Kyber512	AES128	1632	800	768
Kyber768	AES192	2400	1184	1088
Kyber1024	AES256	3168	1568	1568
RSA3072	AES128	384	384	384
RSA15360	AES256	1920	1920	1920

Table 4.1: Kyber performances

While RSA keys are still smaller, Kyber key sizes are of the same order. This is not obvious, because some PQC systems have keys with hundreds of kilobytes, or even in the megabyte range.

Since Kyber is a public key encryption system, it works with public keys for encryption and private keys for decryption. The algorithm works in three different stages.

1. **Key Generation** where a public key and a corresponding private key are created.
2. **Encryption** The encryption phase involves using the receiver's public key to encrypt a message. The process includes generating random polynomials and adding noise to ensure security based on the Learning With Errors (LWE) problem.
3. **Decryption** in this phase the secret key is used to recover the original message from the ciphertext.

For a better understanding of the algorithm a numerical example of each phase of the algorithm is provided. The following variables are expressed using a toy example for a more readable numerical values:

1. Modulo q, since we are operating in on a finite field. In the following example $q = 17$.
2. A polynomial modulo f in the form of $f = x^n + 1$, where n is the degree of the polynomial. The polynomial modulus we will use is $\mathbf{f} = \mathbf{x^4} + \mathbf{1}$. So, by taking a polynomial modulo f, we guarantee that its degree (highest exponent) will be smaller than 4.

From now on all calculations are implicitly done modulo q (on coefficients) and modulo f (on polynomials).

Key Generation

The **private key** of a pair of Kyber keys consists of polynomials with small coefficients. In our example each private key contains two polynomials, for example:

$$\mathbf{s} = (-x^3 - x^2 + x, -x^3 - x)$$

Generating this private key is straightforward.

A Kyber public key consists of two elements. A matrix of random polynomials $\mathbf{A}$ and a vector of polynomials $\mathbf{t}$. Generation of the matrix is fairly simple, we just generate random coefficients and take them modulo q. For example:

$$\mathbf{A} = \begin{pmatrix} 6x^3 + 16x^2 + 16x + 11 & 9x^3 + 4x^2 + 6x + 3 \\ 5x^3 + 3x^2 + 10x + 1 & 6x^3 + x^2 + 9x + 15 \end{pmatrix}$$

To calculate **t** we need an additional vector **e**. This is called error vector and also consists of polynomials with low degree coefficients, exactly like the private key. For example:

$$\mathbf{e} = (x^2, x^2 - x)$$

t is obtained by the following operations:

$$t = \mathbf{As} + \mathbf{e}$$

This makes **t** a vector of polynomials, just like **s** and **e**, by doing the calculations we end up with

$$\mathbf{t} = (16x^3 + 15x^2 + 7{,}10x^3 + 12x^2 + 11x + 6)$$

Now we have

1. Private Key: **s**
2. Public Key: **A,t**

The security of this scheme relies of the fact that recover **s** starting from **(A,t)** is a hard problem, in fact it would require an attacker to solve the learning with error problem. LWE problem is expected to be hard to resolve even for quantum computers and if formally defined as follow:
Given a matrix **A**, a positive integer **q** and a positive real number β find a vector **x** such that:

Proposition 1. $0 \leq |x| \leq \beta$

Proposition 2. $\mathbf{A} \cdot \mathbf{x} \equiv \mathbf{0} mod q$

Encryption

Since kyber is a public key encryption scheme, we can encrypt a message using the public key. The encryption procedure uses an error polynomial vector $\mathbf{e_1}$ and a randomized polynomial vector **r**. These polynomial vectors are generated for every encryption. Additionally, we need an error polynomial $\mathbf{e_2}$. The polynomials within e_1, e_2 and r are random, for example:

$$\mathbf{r} = (-x^3 + x^2, x^3 + x^2 - 1)$$

$$\mathbf{e_1} = (x^2 + x, x^2)$$

$$\mathbf{e_2} = (-x^3 - x^2)$$

To encrypt a message, we have to turn it into a polynomial and it is done by using the message's binary representation. Every bit of the message is used as a coefficient. For example, assume we want to encrypt the number 11.
Eleven has as binary representation 1011, therefore our message encoded as binary polynomial is:

$$\mathbf{m_b} = 1x^3 + 0x^2 + 1x^1 + 1x^0 = x^3 + x + 1$$

Before encryption we have to scale this polynomial. We upscale m_b by multiplying it with $\lfloor q/2 \rfloor$ (the integer closest to q/2). This is done because the coefficients of the polynomial need to be large to make decryption harder. Our example then become:

$$\mathbf{m} = \lfloor q/2 \rfloor \cdot m_b = 8 \cdot m_b = 8x^3 + 8x + 8$$

And now the message is ready to be encrypted using the public key **(A,t)**. The encryption procedure outputs two values (**u**, v):

$$\mathbf{u} = \mathbf{A^t} \cdot \mathbf{r} + \mathbf{e_1}$$

$$v = \mathbf{t}^T \cdot \mathbf{r} + e_2 + m$$

Doing the calculations in our example we obtain

$$\mathbf{u} = (11x^3 + 11x^2 + 10x + 3, 4x^3 + 4x^2 + 13x + 11)$$

$$v = 7x^3 + 6x^2 + 8x + 15$$

The ciphertext obtained from the encryption consists of those two values (u, v).

Decryption

Given the private key **s** and a ciphertext (**u**, v), the decryption process proceeds as follows. First, we have to compute a **noisy result**:

$$\mathbf{m_n} = v - s^T\mathbf{u} \tag{4.1}$$

This result is noisy because of the error vector added during the encryption phase. And now we can see why we needed to scale m by making its coefficient larger. All other terms in 4.1 were chosen to be small. So, the coefficients of m_n are both closer to $\lfloor q/2 \rceil = 8$, implying that the original binary coefficient of m_b was a 1, or closer to 0 implying that the original binary coefficient was 0. We can mitigate this effect by choosing bigger parameter that the one used in this explanation. In the following table we can see some examples of the parameters used in Kyber for a better understanding of its security principles:

Algorithm	**n**	**k**	**q**	$\boldsymbol{\eta_1}$	$\boldsymbol{\eta_2}$	$\boldsymbol{d_u}$	$\boldsymbol{d_v}$	$\boldsymbol{\delta}$
Kyber512	256	2	3329	3	2	10	4	$\boldsymbol{\delta} = 2^{-139}$
Kyber768	256	3	3329	2	2	10	4	$\boldsymbol{\delta} = 2^{-164}$
Kyber1024	256	4	3329	2	2	11	5	$\boldsymbol{\delta} = 2^{-174}$

Table 4.2: Standard Parameters set for Kyber
[23]

Those parameters represents:

1. n: maximum degree of the used polynomials.
2. k: number of polynomials per vector.
3. q: modulus for numbers.
4. η_1, η_2: control how big coefficients of "small" polynomials can be.
5. d_u, d_v: control how much (u,v) get compressed.
6. δ: the probability that a decryption yields a wrong result.

Kyber KEM

Kyber is usually referred to as a Key Encapsulation Mechanism(KEM), not a public key encryption system. A KEM is a cryptographic method that is used to securely exchange encryption keys between parties. It allows a sender to encapsulate a random symmetric key using a receivers's public key, and the receiver can then decapsulate it using its private key to retrieve the symmetric key. The reason for this is technical: Kyber uses a technique to turn the indistinguishability under chosen plaintext attack secure PKE (Public Key Encryption) (see Section 2.1.13) into an Indistinguishability under non-adaptive and adaptive Chosen Ciphertext Attack secure KEM [34] (see Section 2.1.14.)

4.2.2 CRYSTALS-Dilithium

CRYSTALS-Dilithium is a post-quantum digital signature scheme based on lattices. It is based on the hardness of the Module Learning With Errors 2.1.11 and the Closest Vector Problem (CVP) 2.1.12.

Within modern cryptographic communication protocols, Dilithium is one of the main post-quantum schemes for digital signatures, that could substitute non-quantum digital signature schemes, including RSA and those based on elliptic curves, that have been proved to be vulnerable to attacks by a quantum computer.

The algorithm in divided into three main phases:

1. **Key Generation** this phase involves generating public and secret keys. The secret key components are short vectors in the lattice, and the public key is generated using these secret keys and an error term.

2. **Signature Generation**: this phase involves creating a signature that proves knowledge of the secret key without revealing it. This process can be seen as finding a short vector (CVP) and handling noisy lattice points (LWE).

3. **Verification**:The verification process ensures that the signature corresponds to a valid short vector in the lattice.

All three of these phases will be better explained through an example in the following sections.

Before entering in the details of the Dilithium signature scheme, it is useful to establish some notations:

1. q: a prime number

2. $\mathbb{Z}_n$ denotes the set $0, ..., q-1$

3. R_q: Indicates the set of polynomials in X

4. S_η denotes a subset of R_q with coefficients limited from η

5. Let S be a set, $s \leftarrow S$ indicates that s is uniformly sampled from S

6. B_t indicates the subset of R_q consisting of polynomials with t coefficients equal to 0 and the rest to 1

7. H is an hash function with codomain B_t.

Key Generation

In this phase, the private key and the public key are produced, which will be respectively used to generate and verify a signature. A public matrix **A** is uniformly sampled, where each component is a polynomial in the set R_q. The entries of the private key (s_1, s_2) are uniformly sampled from the set S_η. In this part we can notice the instance of the LWE problem. Using the matrix A and the vectors (s_1, s_2), the vector t is derived, which together with A forms the public key of the signature scheme.

It is relevant to note that, since the random matrix A is composed of $k \cdot l$ polynomials in R_q, its representation would be excessively large, making transmission rather costly and therefore making the use of Dilithium less practicable. The adopted solution is to generate A from a smaller-sized seed, thus it is sufficient to transmit only the identified seed within the public key.

Signature

Given a message m, the process of generating a signature σ in Crystals-Dilithium involves the following six steps:

1. **Hash the Message:** Compute a hash of the message m using a cryptographic hash function. Denoted as $H(m)$.

2. **Generate a Random Vector** Generate a random vector y with small, random coefficients. This vector serves as a nonce and ensures the uniqueness of the signature for different messages.

3. **Calculate the Commitment** Compute the commitment $w = A \cdot y$. Use the commitment to compute a hint vector c by hashing w together with the message hash $H(m)$.

4. **Compute the Response** Calculate the response vector $z = y + c \cdot s_1$. Here, c is the hint vector and s_1 is the part of the secret key.

5. **Check Norm**

   Check that the coefficients of z and $w - c \cdot t$ are within certain bounds. If not, discard and start over with a new y.

6. **Create the Signature**

   The signature σ consists of the response z and the hint c.

Verification

In the verification step, it is checked that the commitment c, contained in the signature, matches the output of the hash function computed on the concatenation of the message m and w_1, where w_1 is derived from the signature $\sigma = (z, c)$ and the public key A.
Also in the signature verification process, some optimizations have been introduced that, by working with only the most significant bits, allow for a significant reduction in the size of the digital signature.

Numerical Example

A numerical example is provided for a better visualization of the previous sections (4.2.2, 4.2.2, 4.2.2).

Key Generation

1. **Parameter Selection:** Let's assume small parameters for simplicity. $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

2. **Secret Key Generation:** Generate secret vectors s_1 and s_2 with small coefficients. $s_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad s_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

3. **Public Key Calculation:** $t = A \cdot s_1 + s_2$ $t = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 2 \cdot 1 \\ 3 \cdot 1 + 4 \cdot 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 9 \end{bmatrix}$

Signature Generation Given a message m:

1. **Hash the Message:** Compute $H(m)$. For simplicity, let $H(m) = 2$.

2. **Generate a Random Vector:** Generate a random vector y. Assume $y = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$

3. **Calculate the Commitment:** $w = A \cdot y$ $w = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 2 \cdot (-1) \\ 3 \cdot 1 + 4 \cdot (-1) \end{bmatrix} = \begin{bmatrix} 1 - 2 \\ 3 - 4 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$ Compute $c = H(w||H(m))$.
   Let $H(-1, -1, 2) = 1$.

4. **Compute the Response:** $z = y + c \cdot s_1$ $z = \begin{bmatrix} 1 \\ -1 \end{bmatrix} + 1 \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 + 1 \\ -1 + 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$

5. **Check Norm:** Check z and $w - c \cdot t$ are within bounds. Assume they are.

6. **Create the Signature:** The signature $\sigma = (z, c)$ is $\sigma = \left(\begin{bmatrix} 2 \\ 0 \end{bmatrix}, 1 \right)$

Signature Verification To verify σ on message m:

1. **Compute the Commitment:** $w' = A \cdot z - c \cdot t$ $w' = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 0 \end{bmatrix} - 1 \cdot \begin{bmatrix} 3 \\ 9 \end{bmatrix} = \begin{bmatrix} 1 \cdot 2 + 2 \cdot 0 \\ 3 \cdot 2 + 4 \cdot 0 \end{bmatrix} - \begin{bmatrix} 3 \\ 9 \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \end{bmatrix} - \begin{bmatrix} 3 \\ 9 \end{bmatrix} = \begin{bmatrix} -1 \\ -3 \end{bmatrix}$

2. **Hash the Commitment:** Hash w' together with $H(m)$ to obtain c'. Let $H(-1, -3, 2) = 1$.

3. **Check the Signature:** Verify that $c' = c$ and that z and w' are within bounds.

Since $c' = 1$ and $c = 1$, and assuming all bounds checks pass, the signature is valid.

4.2.3 FALCON

FALCON is a post- quantum digital signature schema based on lattice. This chapter will present the key ideas underlying Falcon and explain how they all fit together based on its original paper [65].

Falcon is based on

1. **q-ary lattices**: characterized by the fact that their points coordinates are integer and "wrapped around" an integer q. If we reduce modulo q the coordinates of a lattice point, the result will still be a point of the lattice

2. **NTRU Lattices** : a lattice-based cryptographic scheme (see Section 4.1) that uses structured polynomials over a ring. NTRU lattices leverage the properties of polynomials and their transformations to achieve efficient key generation, encryption, and decryption operations [53].

3. **Fast Fourier Sampling**: a technique used in lattice-based cryptography, to efficiently sample from discrete Gaussian distributions over lattices[63]. By leveraging fast Fourier sampling, Falcon ensures that cryptographic operations remain efficient and scalable, even with large-scale deployments.

Before entering in the details of the FALCON signature scheme, it is useful to establish some notations:

1. q: called modulus, it is a prime number that defines the size of the finite field over which operations are performed. Larger q provides higher security against attacks but typically requires more computational resources, while smaller q offer better efficiency but could compromise security.

2. n: called dimension, refers to the number of components in the vectors that define the lattice. The dimension n influences the complexity of cryptographic operations and the size of the keys and signatures.

3. T: called trapdoor matrix, refers to a matrix T that possesses a special property: it simplifies operations in one direction (generating a matrix from a vector) but not in the reverse direction (finding the vector from the matrix without knowing T).

4. e: called noise vector, refers to a vector whose components are chosen from a certain distribution, typically a discrete Gaussian distribution. This noise vector is intentionally added to certain computations to ensure the security of cryptographic schemes. Used to achieve protection against various attacks and to introduce randomness becoming resistant to statistical attacks.

5. m: the message to be signed.

6. H: a cryptographic hash function

7. z: the signature performed.

After choosing appropriate parameters this is an high level scheme on how FALCON works:

1. **Key Generation Process** In this phase, the private key and the public key are produced, which will be respectively used to generate and verify a signature.

2. **Signing Process** This phase involves creating a digital signature for a given message using the private key.

3. **Signature Verification** This phase involves checking that a given signature is valid for a specific message and public key. This process ensures that the signature was indeed produced by the holder of the corresponding private key and that the message was not altered.

Those phases will be explained more in detail in the following sections, including also a numerical example for a better understanding of each phase.

Key Generation

This process is based on NTRU lattices [53] and involves the generation of polynomials that define the lattice.

1. **Generate Secret Polynomials**: The first step on the generation of a public and private key-pair is to define two secret polynomials f and g. These are random polynomials with coefficients typically in $\{-1,0,1\}$.

2. **Compute Auxiliary Polynomials**: These polynomials are derived such that they satisfy the NTRU equation

$$fG - gF = q \tag{4.2}$$

3. **Compute the Private Key** The private Key S is is a 2×2 matrix constructed from the polynomials f, g, F and G.

4. **Compute the Public Key** The public key h is derived from the private key polynomials as $h = gf^{-1} \mod q$, where f^{-1} is the modular inverse of f.

Numerical Example A numerical example is useful for a better visualization of the steps explained before. In the following example the value of n will be 12289.

1. Generate Secret Polynomials, called f and g:

$$f(x) = 1 - x - x^3$$

$$g(x) = -1 + x^2 + x^3$$

2. Compute Auxiliary Polynomials F and G:

$$F(x) = 1 + x + x^3$$

$$G(x) = -1 - x^2 + x^3$$

We have to verify the relation $fG - gF = q$:

$$f(x)G(x) = (1 - x - x^3)(-1 - x^2 + x^3)$$

$$= -1 - x^2 + x^3 + x + x^3 + x^5 + x^3 + x^4 + x^6$$

$$g(x)F(x) = (-1 + x^2 + x^3)(1 + x + x^3)$$

$$= -1 - x - x^3 + x^2 + x^3 + x^4 + x^3 + x^4 + x^6$$

In the above calculations, $fG - gF = q$.

3. Construct the **Private key S**:

$$\mathbf{S} = \begin{pmatrix} 1 - x - x^3 & -1 + x^2 + x^3 \\ -1 - x^2 + x^3 & -(1 + x + x^3) \end{pmatrix}$$

4. Compute the **Public Key h**

   $h = gf^{-1} \mod q.$

   Assume $f^{-1}(x) = 1 + x + x^2$

   $h = (-1 + x^2 + x^3)(1 + x + x^2) \mod 12289$

   Simplifying this (omitting intermediate polynomial multiplication steps): $\mathbf{h \equiv x^3 - x + x^2} \mod \mathbf{12289}$ the specific value of the public key h derived from the private key polynomials f and g is determined by how the polynomials are evaluated and combined.

Signature

This process ensures that the signature can be verified later using the corresponding public key and that it authenticates the message's integrity. It is based on the following 5 points:

1. **Message Hashing**: Hash the message m using a cryptographic hash function to obtain μ.
2. **Generate Random Noise**: Generate a small, random polynomial e.
3. **Compute Short Vector s**: Compute s as $s = f \cdot e + G \cdot z \mod q$ Where:
   - f and G are private key polynomials,
   - e is the random noise polynomial,
   - z is another random polynomial used to enhance security.
4. **Construct Signature σ**: The signature σ consists of the short vector s.

Numerical Example Let us continue from the key generation example and use the same variables obtained before.

1. **Compute Short Vector s**: Compute s using the formula: $s = f \cdot e + G \cdot z \mod 12289$ Assume $z(x) = 1 - x^2$ (a random polynomial): $s = (1 - x - x^3)(-1 + x - x^2) + (-1 - x^2 + x^3)(1 - x^2) \mod 12289$ Perform the polynomial multiplication: $f \cdot e = (1 - x - x^3)(-1 + x -$

$x^2) = -1 + x - x^2 + x - x^2 + x^3 - x^3 + x^4 - x^5$ Simplify and reduce modulo q: $f \cdot e = -1 + 2x - 2x^2 + x^4 - x^5 \mod 12289$ Compute $G \cdot z$: $G \cdot z = (-1 - x^2 + x^3)(1 - x^2) = -1 + x^2 - x^3 - x^2 + x^4 + x^5 - x^3 - x^5$ Simplify and reduce modulo q: $G \cdot z = -1 - 2x^3 + x^4 \mod 12289$ Add the results: $s = (-1+2x-2x^2+x^4-x^5)+(-1-2x^3+x^4) \mod 12289$ Simplify: $s = -2 + 2x - 2x^2 - 2x^3 + 2x^4 - x^5 \mod 12289$

2. **Construct Signature σ**:

   - The signature σ is constructed using the computed short vector s: $\sigma = (-2 + 2x - 2x^2 - 2x^3 + 2x^4 - x^5)$

Verification

This process ensures that the signature can be verified later using the corresponding public key and that it authenticates the message's integrity. It's organized in the following steps:

1. **Message Hashing**:

   - The verifier first hashes the message m using the same cryptographic hash function used during the signing phase. This results in a hashed value μ.

2. **Extract Components from Signature**:

   - The signature σ consists of a short vector s derived during the signing. Extract s from σ.

3. **Compute Syndrome**:

   - Using the public key h, compute the syndrome t: $t = \mu - hs \mod q$ Where:
     - h is the public key polynomial derived during key generation,
     - s is the short vector extracted from the signature σ,
     - q is the modulus used in the Falcon lattice.

4. **Verify Syndrome**: Check if t is sufficiently small according to predefined criteria. Typically, this involves ensuring each component of t falls within a specific range or bound.

5. **Conclusion**:

   - If t meets the criteria (i.e., all components are within bounds), the signature σ is valid for the message m under the public key h.
   - If any component of t exceeds the bound, the signature σ is considered invalid.

Numerical Example The numerical example concludes all of the previous example showing us how the signature can be verified.

1. **Compute Syndrome t**:

   - Compute hs: $hs = (x^3 - x + x^2)(0, 1, -1, 0) = (0, x^3 - x + x^2, -x^3 + x - x^2, 0)$
   - Compute t: $t = \mu - hs \mod 12289$ Substitute $\mu = (1, 0, 1, -1)$: $t = (1, 0, 1, -1) - (0, x^3 - x + x^2, -x^3 + x - x^2, 0) \mod 12289$ Simplify to find t.

2. **Verify Syndrome**: Check if each component of t is within the acceptable range specified by the protocol. For instance, ensure each component is sufficiently small in relation to the modulus 12289.

3. **Conclusion**:

   - If all components of t are within bounds, the signature $\sigma = (0, 1, -1, 0)$ is valid for the message "Hello" under the public key $h = x^3 - x + x^2 \mod 12289$.
   - If any component of t exceeds the bound, the signature would be deemed invalid for the given message and public key.

4.2.4 SPHINCS+

SPHINCS+ (SPHINCS-256) is a post-quantum digital signature algorithm that relies on the security of cryptographic hash functions, it uses trees whose nodes are Merkle trees (see Section 4.2) and one-time signature (OTS) schemes (see Section 4.2) to provide a post-quantum secure digital signature.

In Figure 4.3 we can see an example of how a Merkle tree works, starting from 8 secret keys $s_1..s_8$

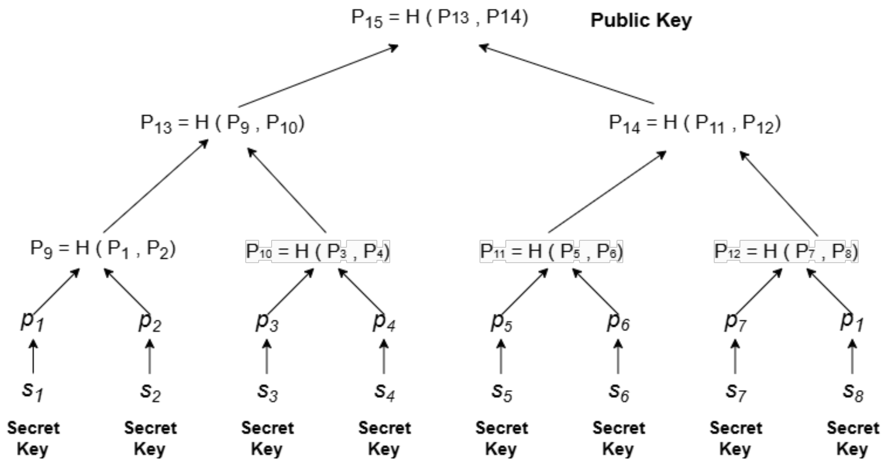


Figure 4.3: Signature in 8-time Merkle hash tree

The leaves are made of a couple (s_i, p_i) where p_i is computer starting from the secret keys through a one time signature schema.

The layers are then chained 2 by 2 and then hashed to generate the new layer until we arrive to the root (P_{15} in 4.3), P_{15} is the **Public Key** .

To sign a message (m_1) the signer uses the couple (S_1, P_1) so the message is signed with the secret key S_1.

The signature of m_1 is $(sign(m_1, S_1), P_1, P_10, P_14)$ so a verifier needs also P_10 and P_14 to verify a signature.

The verifier receives $sign((m_1, S_1), P_2, P_10, P_14)$ and knows P_{15}.

He computes P'_9 by hashing p_1 and p_2, p'_{13} by hashing p'_9 and p_{10} and p'_{15} by hashing p'_{13} and p_{14} (see Figure 4.4).

The verification is done by comparing p'_{15} with P_{15}, if the signature calculated is the same as the top leaf of the tree the signature is valid, otherwise it would be rejected.

All the nodes that needs to be used from the verifier compose an **Authentication Path** as shown in figure 4.4

With this scheme is possible to **sign more messages with the same public key** (2^n, where n is the number of levels in the tree), and the public key is short, since it is just an hash output. The problem of this scheme is that computing the public key requires computing and storing 2^n one time signature keys (s_i)and each signature contains n public keys.

The Merkle tree also do not provides a **stateful scheme** because signing a message requires to keep state of the already used keys, making sure they

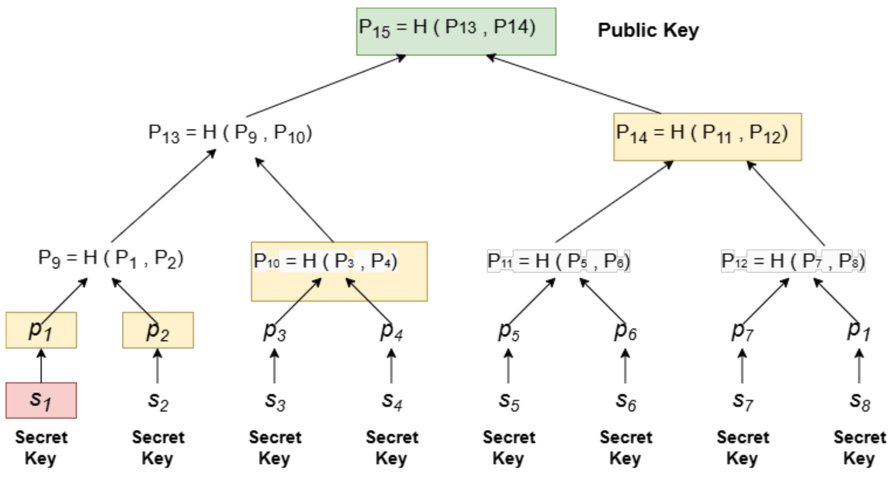


Figure 4.4: Authentication Path for a signature in 8-time Merkle hash tree

are never reused.

Stateful hash-based signatures

Stateful hash-based signatures are a class of digital signature schemes that leverage hash functions to provide secure signatures. The security features introduced by this schema are:

1. **Post Quantum Resistance**: Stateful hash-based signatures are considered secure against attacks from quantum computers

2. **Forward Security**: Since each OTS key pair is used only once, compromising one key does not affect the security of other keys or past signatures.

3. **Minimal Security Assumptions**: The security of these schemes relies mainly on the strength of the hash function, which is a well-understood cryptographic primitive.

On the other hand this scheme can be broken by reusing the keys.

HyperTree

The aim of SPINCS+ is to: Achieve moderate signature time and size and get rid of any state in the nodes.

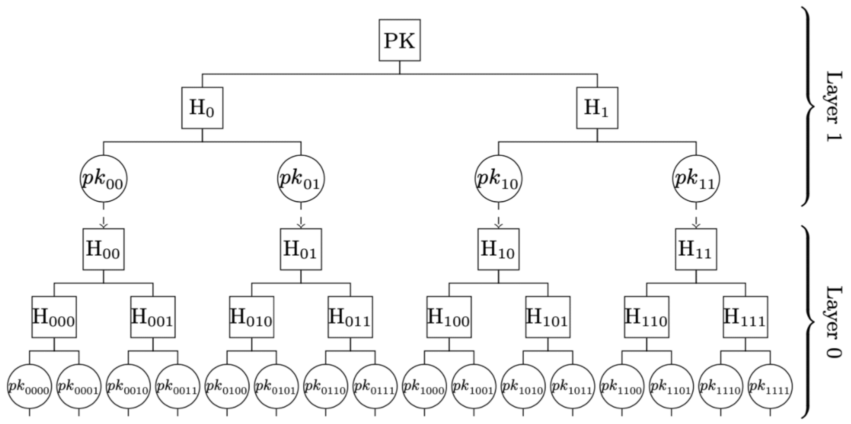


Figure 4.5: HyperTree of height 4 which can be seen as a tree with merklee trees on the nodes

This is done by using a structure in which every leaf in a tree structure is another tree (see Figure 4.5).

In SPHINCS, each leaf has an address, which contains: its layer in the SPHINCS hypertree, the number of its Merkle's tree in the layer, its position in the Merkle's tree.

Moreover, in addiction to OTS, SPINCS+ use **Few-Time Signatures (FTS)** like FORS (Forest of Random Subsets) (see [7]) which can sign a limited number of messages securely.

SPHINCS+ have the pro of being based on hash cryptography, which is a consolidated field of study, so it allows a conservative choice unrelated to lattice-based cryptography but, on the other hand, the scheme significantly affects the performance of the protocols that uses it, both in computation time and in size of signatures [25].

Key Generation

The key generation process in SPHINCS+ involves several steps to ensure the creation of a secure and efficient key pair.

1. **Parameter Selection** Choose parameters for the scheme, including the number of layers in the hypertree, the height of each Merkle tree, and the parameters for the OTS and FTS schemes

2. **Seed Generation** Generate a random seed for the secret key. This

seed will be used to derive all other key pairs and parameters in the scheme.

3. **OTS Key Pair Generation** For each leaf node in the lowest level of the hypertree, generate a OTS key pair using the seed and the hash function.

4. **FORS Key Pair Generation** Generate FORS key pairs for signing.

5. **First-Level Merkle Tree Construction**: Construct a Merkle tree using the WOTS public keys as leaf nodes. Compute the parent nodes by hashing the child nodes until the root of the tree is obtained.

6. **Upper-Level Merkle Trees Construction**: Use the roots of the lower-level Merkle trees as leaf nodes to construct the Merkle trees for the next level. Repeat this process up to the root of the top-level Merkle tree.

7. **Public Key Generation**:The public key of the SPHINCS+ scheme is the root of the top-level Merkle tree.

8. **Private Key Storage**: The private key consists of the initial seed and the parameters used to derive the OTS and FORS key pairs.

Signature

The signature generation process in SPHINCS+ involves several steps upon the key structures generated during the key generation phase.

1. **Message Digest**: Compute a hash of the message m to be signed. This hash will be used as the input for the signature generation process.

2. **Randomness Generation**: Generate a random value,r, which is unique for each signature. This ensures that even if the same message is signed multiple times, the signatures will be different.

3. **Index Selection**:Determine the indices for the OTS and FORS key pairs that will be used in the signature by selecting specific leaves in the tree structure.

4. **FORS Signature**: Sign the message hash with the FORS scheme

5. **OTS Signature**:Sign the output of the FORS signature with the OTS scheme
6. **Authentication Paths**: For each level in the hypertree, include the necessary authentication paths to authenticate the WOTS public keys. Compute the paths from the selected WOTS leaf nodes up to the root of the top-level Merkle tree. These paths allow the verifier to reconstruct the Merkle tree roots and ensure the integrity of the public keys
7. **Constructing the Signature**: Combine all components into the final signature, the random value r, the FORS and OTS signatures and authentication paths, the authentication paths for all levels in the Hypertree.

Verification

The SPHINCS+ signature verification process ensures the authenticity and integrity of a signed message using a combination of hash-based cryptographic primitives and hierarchical Merkle trees.

1. **Extract Components from Signature**: Extract the random value r, the FORS signature $SIGN_{FORS}$ with its authentication path, the OTS signature $SIGN_{OTS}$ with its authentication path, and the authentication paths for all levels in the hypertree.
2. **Recompute Message Digest**: Compute the hash of the message $H(m)$
3. **Verify FORS Signature**: Using the FORS public key derived from the signature.
4. **Verify OTS Signature** Using the OTS public key derived from the FORS signature.
5. **Verify Authentication Paths**: For each level in the hypertree, verify the authentication paths. Starting from the lowest-level Merkle tree and move up to the top-level Merkle tree to ensure that the public keys at each level are correctly integrated into the Merkle tree structure
6. **Final Verification**: Ensure that the root of the top-level Merkle tree matches the public key of the SPHINCS+ scheme. If all checks pass, the signature is valid.

Chapter 5

Post-Quantum TLS

Transport Layer Security (TLS) (see Section 3.4) is one of the most important security protocols of the Internet. This fundamental protocol protects the confidentiality, authenticity, and integrity of individual communication channels in potentially malicious network environments. TLS is one of the most widely used tools for securing application protocols. Like any other protocol that uses public key cryptography, the security of TLS is threatened by future quantum attackers. This is because quantum computers can efficiently break the underlying hardness assumptions of the public key cryptography on which the TLS handshake protocol is currently based, namely the integer factorization and discrete logarithm problems (see Section 2.5.5). Thus, future adversaries with such powerful machines can undermine the security of TLS by forging signatures to falsely authenticate themselves to users, and by deriving encryption keys to read secret messages, even from past TLS sessions.

In this chapter, we will discuss the integration of two specific post-quantum algorithms: Dilithium for digital signatures and Kyber for key encapsulation mechanisms (KEM). These algorithms were chosen due to their robustness and efficiency, as evidenced by their performance in the NIST post-quantum cryptography standardization process [9] and in many research papers such as [57] and [60]. Both the cited papers will be two keystones for the following work.

The modifications made to the TLS 1.3 protocol involve several aspects, such as

1. Key Exchange: Integration of Kyber KEM to secure the key exchange process against quantum attacks.

2. Digital Signatures:Incorporation of Dilithium to provide quantum-resistant digital signatures.

3. Encryption: Ensuring that encryption mechanisms within the protocol are compatible with post-quantum standards.

4. Changes to digital certificates: adapting the X.509 certificates to the proposed changes.

There are also other implementation of the Post Quantum TLS in different programming languages on Github such as [61], done in Rust or the liboqs repository [58] that provides implementation of the main components both for the TLS algorithm and the post quantum algorithms in C language.

By implementing these changes, our primary goal is to enhance the security of the TLS protocol against potential quantum computer threats, also offering a secure communications protocol in the future.

Through this work, our objective is to contribute to ongoing efforts in cryptographic research to develop and deploy quantum-resistant security protocols, ensuring the confidentiality, integrity, and authenticity of communications in the quantum era.

5.0.1 Post-quantum TLS handshake

In this section we will explore the proposed changes to the standard TLS handshake architecture. The proposed changes are intended not only to **integrate post-quantum algorithms** into the protocol but also **to maintain retro-compatibility** to the actual TLS.

The retro-compatibility property ensures us the fact that the actual structure of the TLS architecture can be maintained without the need of major changes such as adding fields to the X.509 certificate (see Figure 3.1) with the whole post-quantum signature of the certificate.

Since the goal is to make the TLS protocol post-quantum resistant, the client and the server during the handshake procedure will agree to use algorithms that guarantee a post-quantum level of security.

Changes to the Digital Certificate

The first change to the previously discussed protocol is related to the Digital Certificate used for the authentication of the client and server that are willing to communicate in a secure way.

The differences are in the **extension field**: since we now are also giving the possibility to use PQ algorithms, we have to add to the certificate the post-quantum public key of the certificate and the post-quantum digital signature of the certificate into the Certificate.

Adding those fields into the extensions section of the X.509 allow us to use **the same structure** of the certificate, without the need of radically changing its fields, granting us the retro-compatibility. The new post-quantum field will also be used in the verification phase by the Certificate Authority (see section 3.1.1).

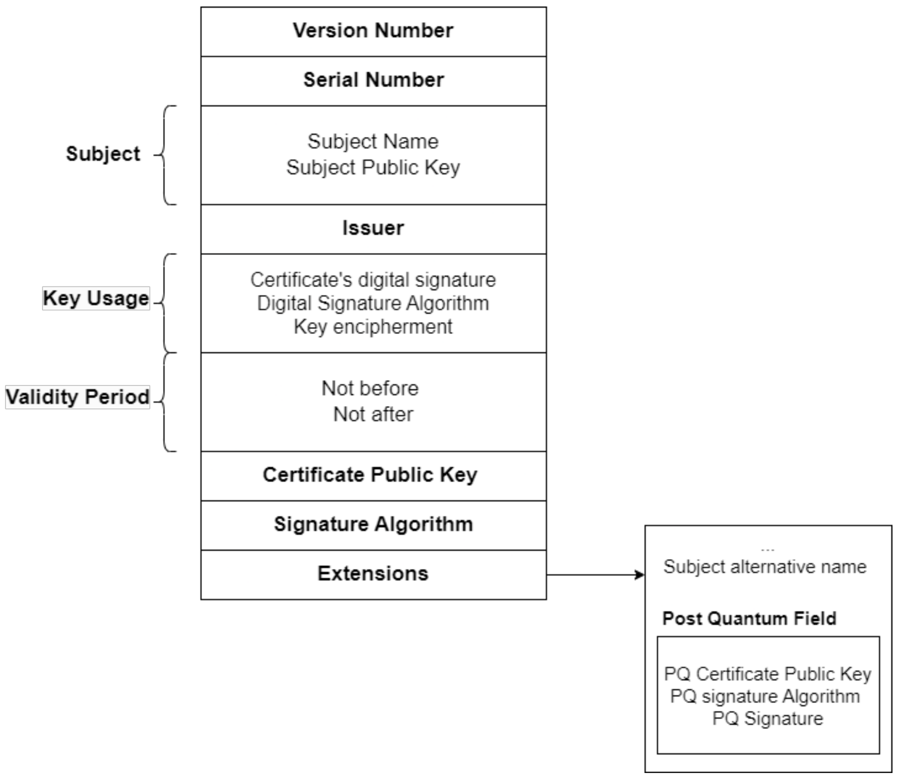


Figure 5.1: Structure of a modified X.509 Certificate

Modified TLS Handshake

After the discussion on how the Digital Certificate has to be modified, we are now going to discuss the **Post-Quantum TLS handshake** with reference to figure 5.2 as a guide.

The handshake will be described in 6 steps, listed below and also present in Figure 5.2.

1. At the beginning of the connection we have a client and a server that are willing to communicate, each one will produce a random integer number referred respectively as **server random** for the server and **client random** for the client.

   In a preliminary phase correspondent to the **Client Hello** and **Server Hello** messages (see section 3.4) the client and the server that are willing to communicate will exchange some preliminary information, such as the security level they want to achieve in the communication, the ciphersuite they will use, and they will exchange their random numbers.

2. The communications starts with the server sending their certificate to the client, giving the client a way to verify that the server is actually who is supposed to be.

3. When the client receives the certificate, it proceeds to verify it, checking the Certification Authority up to the root until it can verify its validity. If this step is not performed correctly, the handshake stops here and the connection is closed.

   If the Certificate is proved to be valid, the client retrieves from the extensions field of it the **post-quantum Public key** of the server, and the **Post-Quantum digital signature** of the certificate.

   The Digital signature will be validate using the post-quantum key previously retrieved from the certificate in the new PQ section added in the extension field (see Figure 5.1). The keys are generated from the client through the specific functions of the chosen signature algorithm. See [33] for the key generation function. If the signature is valid, the connection goes on; otherwise, the connection will be closed.

4. If the signature is proved to be valid, the client now has to achieve the **Proof of Possession** (PoP) property, that means that he wants a proof that the server posses the private key corresponding to the public one we took from its certificate. To obtain it the client will generate a random string using a pseudo-random function, called **pre-master secret** because it is the first step to obtain the master secret that will make the communication safe. After that, the encryption is done using the Kyber Public key of the server, previously retrieved from the

server's certificate, with the kyber encryption function [48].

The result of this encryption will be a ciphertext representing the random string, encrypted with the kyber public key of the server.

5. At this point the Key Encapsulation Mechanism offered by kyber is used to encapsulate the ciphertext of the pre-master secret. This step is done to make it resistant to sniffing attacks. Even if an attacker sniffs the packet with the ciphertext will still not be able to decrypt it.

   The encapsulated packet is sent to the server that will decapsulate it and will us its Kyber private key to decrypt it and to obtain the Pre Master secret. The encaps and decaps function are available in the kyber Github [48].

   If the pre master secret received is different from the one generated from the client, the connection will occur in an error and will be stopped; indeed, it is possible that an attacker sent a packet trying to guess the pre master secret. Otherwise, the server and the client now have access to the same **pre-master secret** and will use it to build the **master secret**.

   The obtained master secret will be the shared secret that makes the connection secure.

6. The client and the server will now generate the Master Secret using a Key Derivation Function (KDF) that takes as input the concatenation of the client random, the server random and the ciphersuite string choosed at the start of the communication. Since all the values in it are shared between the client and the server, they will produce the same key.

   The KDF function is then performed not only once but a variable number depending of the KDF used.

At the end of this process the client and the server will have a secret shared key and from now on the communication will continue using the shared key as the encryption parameter. This process allows the creation of a shared secret key for two unknown entity, and by using post-quantum algorithm will achieve also the post-quantum protection for the protocol.

5.0.2 Mutual Authentication Post-Quantum TLS

As previously said in chapter 3.4, while the authentication of the server is a mandatory requisite, TLS can also supports the mutual authentication between client and server. This scheme can be useful in many cases, such as the implementation of an embedded device that needs to automatically connect to a server. The scheme presented before in figure **??** can cover this eventuality with few modifications.

1. At the beginning of the connection we have a client and a server that are willing to communicate, each one will produce a random integer number referred respectively as **server random** for the server and **client random** for the client.

   During the **Client Hello** and **Server Hello** messages (see Section3.4) the client and the server will exchange some preliminary information: the security level they want to achieve in the communication, the ciphersuite they will use, and the exchange of their random numbers.

   Later on the communication those randoms ensures that, if the server or the client changes during the communication, the connection does not proceed since the new receiver (or sender) will have a different random number.

2. When the client verifies the validity of the server, the connection won't proceed as explained before because the mutual authentication has to be performed, so not only the server has to send its certificate but also the client so there will be another Send Certificate message.

3. The certificate received from the client will be validated in a similar way as the precedent scheme (Figure 5.2). When the server receives the certificate, it proceeds to verify it, checking the Certification Authority up to the root until it can verify its validity.

   If this verification is not performed correctly, the handshake stops here, and the connection is closed without exchanging other data and returning an error message. Otherwise, the connection continues with the server and the client mutually authenticated.

4. If the certificate is confirmed to be valid, the client will extract the server's post-quantum public key and the post-quantum digital signature of the certificate from its extensions field. The digital signature is

verified using the post-quantum key previously obtained from the certificate in the new PQ section included in the extension field (see Figure ??).

Upon successful verification of the signature, the client must ensure the Proof of Possession (PoP) property. The client generates a random string using a pseudo-random function (the pre-master secret). The client then encrypts the pre-master secret using the server's Kyber public key, previously extracted from the server's certificate, obtaining a ciphertext.

5. At this stage, the Key Encapsulation Mechanism provided by Kyber is utilized to encapsulate the ciphertext of the pre-master secret, ensuring resistance to eavesdropping attacks.

   The encapsulated packet is then sent to the server, which will decapsulate it and use its Kyber private key to decrypt it and retrieve the pre-master secret.

6. If the pre-master secret received by the server differs from the one generated by the client, the connection will result in an error and will be stopped, as it indicates a possible attack where an attacker sent a guessing packet. If the pre-master secret matches, both the server and the client will use it to construct the master secret, which will be the shared secret securing the connection.

7. The client and server will now derive the Master Secret using a Key Derivation Function (KDF) applied multiple times to the same input: a concatenation of the client random and the server random, and the ciphersuite string chosen at the start of the communication.

From now on both client and server have a common shared secret, so the connection will continue in a secure way, since the data exchanged will be encrypted using the master key.

5.1 Post-Quantum TLS measurements

In this section contains the performance analysis of a post-quantum TLS scheme not only with the pair Kyber/Dilithium chosen as post-quantum algorithms , but also with the other algorithm chosen by the NIST. We also compare the values obtained with the one provided by the standard TLS with RSA/ECDHE and ECDSA/ECDHE, currently the most common algorithms used in standard TLS.

The performance evaluation in Figure 5.4 refers to a post- quantum TLS scheme implemented on resource-constrained embedded devices. The evaluation is performed on the ARM Cortex-M4 embedded platform NUCLEO-F439ZI, that provides 180 MHz clock rate, 2MB Flash Memory and 256KB SRAM. With regarding to TLS design and open-source implementation,the project is developed using a cryptographic library named liboqs [51], that has collected implementations of PQC algorithms.

In figure 5.4 the notation column indicates which couple of algorithms are used for the PQ handshake, where Dil stands for Dilithium, Kyb for Kyber, Falc for Falcon and Sph for Sphincs+. Moreover, there is a number associated with the algorithms that refers to the security level achieved. For example, Dil3 means that we are using the Dilithium algorithm as a digital signature algorithm that provides us the security level 3. The Static Usage and the .bss Usage columns refers to memory requirements needed. The first one is telling us how much static memory is needed to perform the algorithms, while the second one is telling us that an additional memory section (the bss section) is needed since there are some variables in the code that are declared but not explicitly initialized.

The “Communication Size” column tells us the byte size of all messages exchanged during the TLS handshake, which is the sum of all the bytes that a peer has sent plus the sum of all the bytes that the peer has received. The last column indicates the time needed to perform the handshake and is detailed with the time needed by the client and the time needed by the server.

The measurements collected from the client differ from the measurements collected from the server. That is because although the TLS handshake authentication related operations are similar between the two peers (1 “Sign” and 2 “Verify” operations for each), this is not the case for KEM related operations where the client executes 1 “Key Generate” and 1 “encapsulate” operation, while the server executes 1 “decaps” operation.

We can also verify that KEM combinations with Dilithium perform much better than KEM combinations with Falcon. This is due to the fact that Falcon has extremely fast “Verify” but slow “Sign” operations [57]. We also see that using sphincs+ leads to an extremely slow handshake, with over 66 seconds of run-time, which is an expected result since SPHINCS+ is the only algorithm hash-based, while all the other ones are lattice-based.

In terms of communication size, we can notice that KEM combinations with Dilithium generally perform much better in terms of speed, compared to KEM combinations with Falcon; however, it can be observed that Dil2+Kyb1 requires more than twice the bandwidth than Falc1+Kyb1. The same applies to higher security levels.

Compared to the classical TLS (see Figure5.5), PQ TLS introduces a larger overhead in terms of network traffic due to the excessive communication size. Dil2+Kyb1 uses 6.26 times more bandwidth than ECDSA+ECDHE, while Falc1+Kyb1, having a lower overhead, consumes only 2.9 times more bandwidth than ECDSA+ECDHE.

The data and .bss segments play a crucial role in the PQC algorithm integration in embedded systems.

Algorithms introducing large artifact sizes or having an implementation that requires a lot of Stack memory, may eventually be impossible to be integrated in a memory-constrained device, and that is the reason why Dil5 is not included, since it has too large memory requirements.

Given an average embedded system evaluation board, which has 192 KB usable RAM, Dil2+Kyb1 consumes approximately 25 % of the total available memory, while on higher security levels, Dil3+Kyb5 uses $\approx$ 35%. Combinations with Falcon generally use less memory: Falc1+Kyb1 consumes $\approx$ 22% of the total available memory and Falc5+Kyb5 uses $\approx$ 43%. On the other hand, Sph1s-Kyb1 uses merely 800 bytes of RAM.

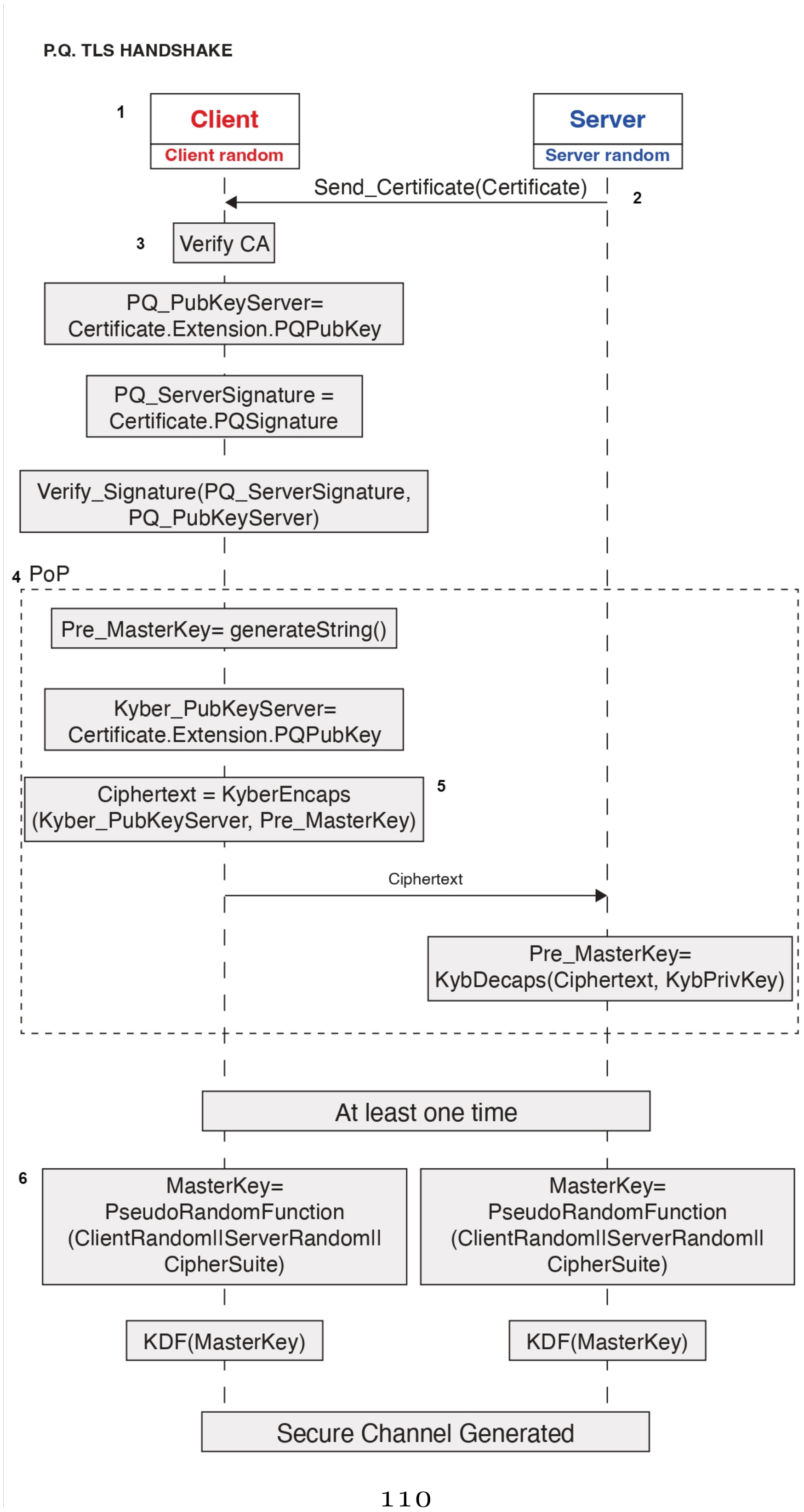

P.Q. TLS HANDSHAKE
1
Client
Client random
Server
Server random
Send_Certificate(Certificate)
2
3
Verify CA
PQ_PubKeyServer= Certificate.Extension.PQPubKey
PQ_ServerSignature = Certificate.PQSignature
Verify_Signature(PQ_ServerSignature, PQ_PubKeyServer)
4 PoP
Pre_MasterKey= generateString()
Kyber_PubKeyServer= Certificate.Extension.PQPubKey
Ciphertext = KyberEncaps (Kyber_PubKeyServer, Pre_MasterKey)
5
Ciphertext
Pre_MasterKey= KybDecaps(Ciphertext, KybPrivKey)
At least one time
6
MasterKey= PseudoRandomFunction (ClientRandom||ServerRandom|| CipherSuite)
MasterKey= PseudoRandomFunction (ClientRandom||ServerRandom|| CipherSuite)
KDF(MasterKey)
KDF(MasterKey)
Secure Channel Generated

Figure 5.2: Post-Quantum TLS Handshake

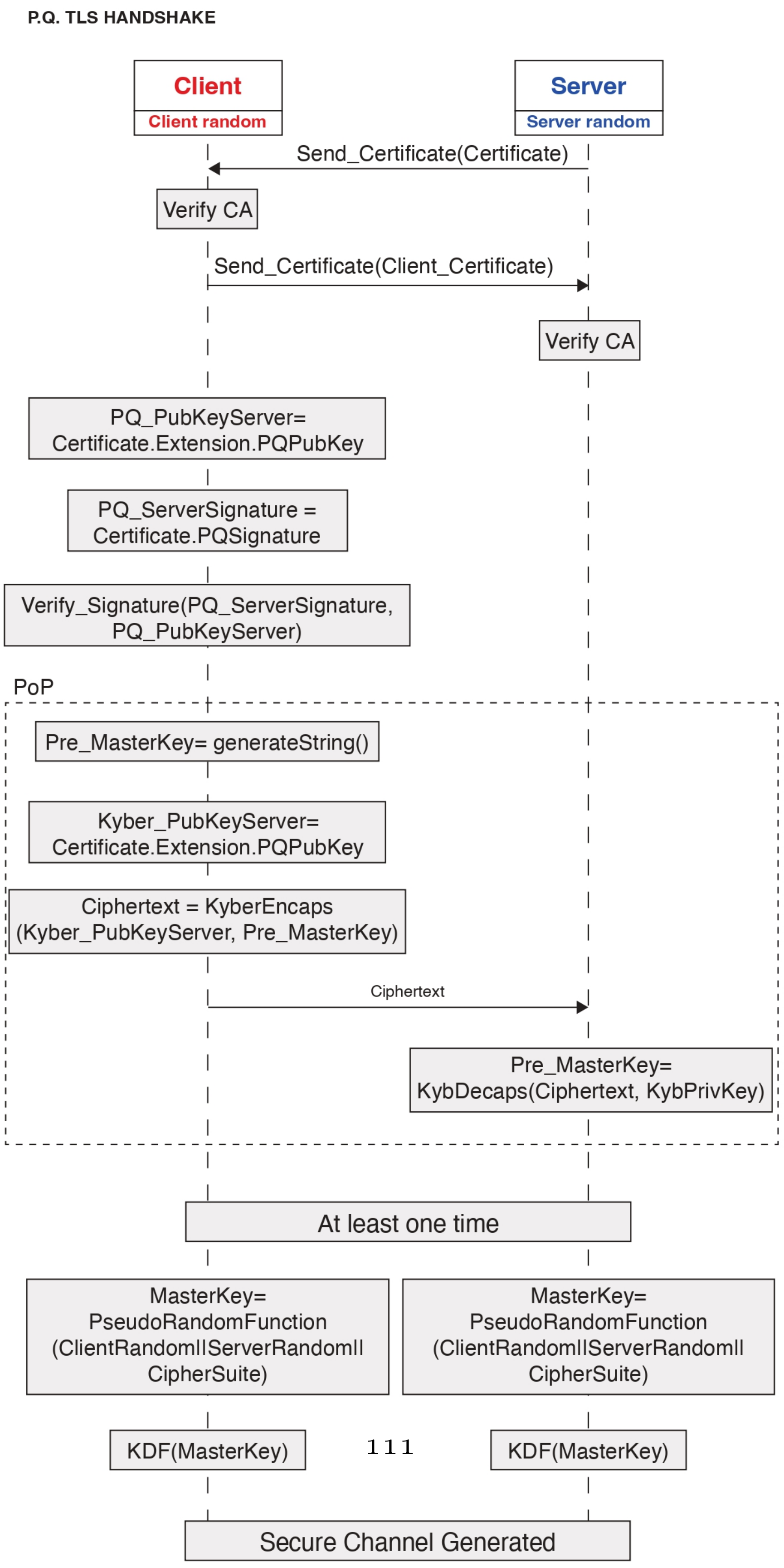
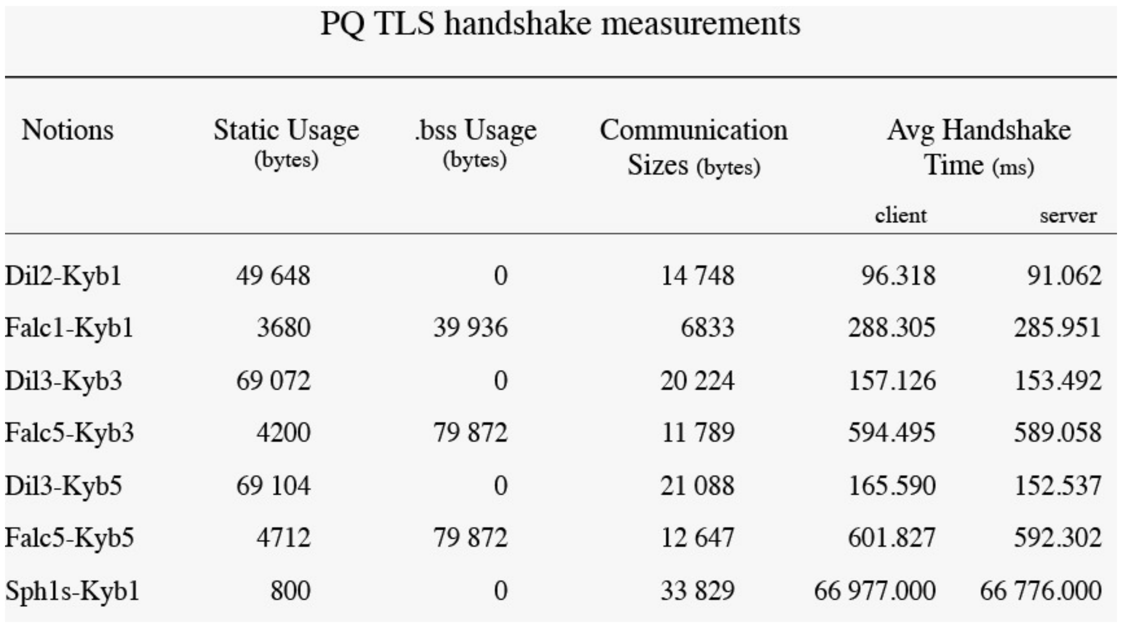

P.Q. TLS HANDSHAKE
Client
Client random
Server
Server random
Send_Certificate(Certificate)
Verify CA
Send_Certificate(Client_Certificate)
Verify CA
PQ_PubKeyServer= Certificate.Extension.PQPubKey
PQ_ServerSignature = Certificate.PQSignature
Verify_Signature(PQ_ServerSignature, PQ_PubKeyServer)
PoP
Pre_MasterKey= generateString()
Kyber_PubKeyServer= Certificate.Extension.PQPubKey
Ciphertext = KyberEncaps (Kyber_PubKeyServer, Pre_MasterKey)
Ciphertext
Pre_MasterKey= KybDecaps(Ciphertext, KybPrivKey)
At least one time
MasterKey= PseudoRandomFunction (ClientRandom||ServerRandom|| CipherSuite)
MasterKey= PseudoRandomFunction (ClientRandom||ServerRandom|| CipherSuite)
KDF(MasterKey)
KDF(MasterKey)
Secure Channel Generated

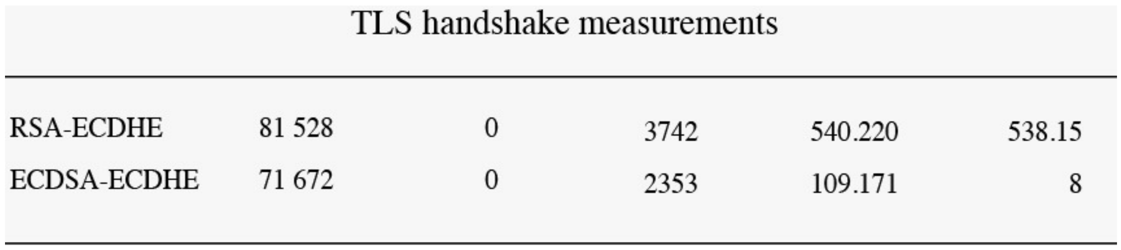
Figure 5.3: Post-Quantum TLS Handshake with Mutual Authentication



PQ TLS handshake measurements

Notions	Static Usage (bytes)	.bss Usage (bytes)	Communication Sizes (bytes)	Avg Handshake Time (ms)	
				client	server
Dil2-Kyb1	49 648	0	14 748	96.318	91.062
Falc1-Kyb1	3680	39 936	6833	288.305	285.951
Dil3-Kyb3	69 072	0	20 224	157.126	153.492
Falc5-Kyb3	4200	79 872	11 789	594.495	589.058
Dil3-Kyb5	69 104	0	21 088	165.590	152.537
Falc5-Kyb5	4712	79 872	12 647	601.827	592.302
Sph1s-Kyb1	800	0	33 829	66 977.000	66 776.000

Figure 5.4: Post-Quantum TLS Handshake measurements, this Figure refers to [57]



TLS handshake measurements

RSA-ECDHE	81 528	0	3742	540.220	538.15
ECDSA-ECDHE	71 672	0	2353	109.171	8

Figure 5.5: Standard TLS Handshake measurements from [57]

Chapter 6

Conclusion

In this thesis, we discuss a way to implement post-quantum algorithms into the TLS handshake scheme. In particular, this work is divided in two main parts. The first one is focused on as a state-of the art on the main cryptographic algorithms (see Chapter2) and the security protocols that use them 3, highlighting the TLS protocol with a detailed description of how its handshake phase works 3.4. The second part explores the post-quantum cryptographic environment (see Chapter 4), explaining the main post-quantum security algorithms providing also numerical examples for each.

This part also introduces the main organizations that leads the transition in the post-quantum world such as the NIST. The last chapter of the thesis is a scheme that describes how two of the main post-quantum algorithms (CRYSTALS-Dilithium for digital signatures and CRYSTALS-Kyber for the Key Encapsulation Mechanism) can be added into the actual TLS handshake process 5.

This work was done considering the need to maintain compatibility with the current handshake process; this was taken into account by modifying some parts of it, such as the X.509 extension field, avoiding structural changes.

This thesis not only dicuss the advances in integrating post-quantum cryptographic solutions into security protocols, but also highlights the significant threat posed by quantum computing to current cryptographic standards.

This work stands as a clear indicator of the present and growing quantum threat, emphasizing the critical need for the cybersecurity community to adopt and implement post-quantum solutions proactively. It is imperative to be ahead with respect to these emerging threats, to ensure the continued protection of digital communications and sensitive data in an increasingly complex and quantum-capable world. As we move towards a future where quantum computing becomes more prevalent, the proactive adoption of these solutions is not just a necessity but a critical step towards safeguarding our digital infrastructure against new threats.

Bibliography

[1] Elliptic curve digital signature algorithm. *HYPR*, 2022.

[2] Ako Muhammad Abdullah. Advanced encryption standard (aes) algorithm to encrypt and decrypt data. *ResearchGate*, 2017.

[3] Paul C. Kocher Alan O. Freier, Philip Karlton. The secure sockets layer (ssl) protocol version 3.0. *Internet Engineering Task Force (IETF)*, 2011.

[4] Minas Dasygenis Argyrios Sideris, Theodora Sanida. High throughput implementation of the keccak hash function using the nios-ii processor. *MDPI*, 2020.

[5] Rahul Awati. Electronic code book (ecb). *TechTarget*, 2021.

[6] Daniel J. Bernstein. Grover vs. mceliece. *The University of Illinois*, 2010.

[7] Leo Breiman. Random forests. *University of California*, 2001.

[8] William C.Barker. Nist sp 800-59, guidelines for identifying an information system as a national security system. *NIST Special Pubblication 800-59*, 2003.

[9] NIST Computer Security Resource Center. Post-quantum cryptography | csrc. *csrc.nist.gov*, 2022.

[10] NIST Computer Security Resource Center. Three draft fips for post-quantum cryptography | csrc. *csrc.nist.gov*, 2023.

[11] Michael Cobb. What is triple des and why is it being disallowed? *TechTarget*, 2023.

[12] IBM Corporation. The tls 1.3 protocol. *IBM*, 2024.

[13] Robert B Davies. Exclusive or (xor) and hardware random number generators. *Robertnz.net*, 2002.

[14] DevX. Cipher block chaining. *DevX*, 2023.

[15] Scott Vansto Don Johnson, Alfred Menezes. The elliptic curve digital signature algorithm (ecdsa). *Certicom*, 2001.

[16] Fabio Donatantonio. Il cifrario di ceare in php. *Donatantionio.net*, 2010.

[17] Nicky Mouha Elaine Barker. Recommendation for the triple data encryption algorithm (tdea) block cipher. *NIST computer security resource center*, 2017.

[18] ETSI. Etsi 2nd quantum-safe crypto workshop in partnership with the iqc. *ETSI*, 2014.

[19] EndroyonoAchmad Affandi Farah Jihan Aufa. Security system analysis in combination method: Rsa encryption and digital signature algorithm. *International Conference on Science and Technology (ICST)*, 2018.

[20] Eric Gershon. New qubit control bodes well for future of quantum computing. *PHYS.org*, 2013.

[21] Jeff Gilchrist. *Encyclopedia of Information Systems.* 2003.

[22] Ginni. What are the role of s-boxes in des. *tutorialspoint*, 2022.

[23] Ruben Gonzalez. Kyber - how does it work? *Approechable Cryptography*, 2021.

[24] J. Orlin Grabbe. The des algorithm illustrated. *Laissez Faire City Times, Vol 2, No. 28.*

[25] M. C. MOLTENI L. NAVA A. GRINGIANI G. GRECO. Integration of pqc in tls protocol for iot devices. *Qubip.eu*, 2024.

[26] Amir Herzberg. Applied introduction to cryptography and cybersecurity. *ResearchGate*, 2023.

[27] Yonghwi Kwon Hyunwoo Lee, Doowon Kim. Tls 1.3 in practice: How tls 1.3 contributes to the internet. *WWW '21: The Web Conference 2021*, 2021.

[28] IBM. Grover's algorithm. *IBM quantum Learning*, 2023.

[29] Raymond G. Kammer. Data encryption standard (des). *FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION*, 1999.

[30] krishna693rah. Triple des (3des). *geeksgorgeeks*, 2024.

[31] JOSH LAKE. the sha-2 algorithm. *comparitech*, 2023.

[32] Philip Leong. Implementation of an fpga based accelerator for virtual private networks. *researchgate.net*, 2002.

[33] Vadim Lyubashevsky. dilithium. `https://github.com/itzmeanjan/dilithium`, 2023.

[34] Varun Maram and Keita Xagawa. Post-quantum anonymity of kyber. *NTT Social Informatics Laboratories, Department of Computer Science, ETH Zurich*, 2022.

[35] Maschen. Own work. *https://commons.wikimedia.org/w/index.php?curid=25757569*, 2023.

[36] Peter L Montgomery. A survey of modern integer factorization algorithms. *Quarterly*, 1997.

[37] Duncan Napier. The ssh protocol. *Enterprise Operations Management*, 2001.

[38] NIST. Submission requirements and evaluation criteria ,for the post-quantum cryptography standardization process. *NIST*, 2016.

[39] NIST. Module-lattice-based key-encapsulation mechanism standard. *FIPS 203*, 2023.

[40] NIST. National institute of standards and technology. Technical report, NIST, 2024.

[41] National Institute of Standards and Technology (NIST). Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. *Special Publication 800-131A*, 2023.

[42] National Institute of Standards and Technology. Sha-3 standard: Permutation-based hash and extendable-output functions. *NIST*, 2015.

[43] W. Simpson P. Karn, P. Metzger. The esp des-cbc transform. *IETF*, 1995.

[44] Pramod Pandya. Advanced data encryption. *Cyber Security and IT Infrastructure Protection*, 2014.

[45] James A. St. Pierre. Fips 197. *Federal Information Processing Standards Publication*, 2023.

[46] preetikintali. What is p-box in cryptography. *geeksforgeeks*, 2023.

[47] Saleem Raza. Ethereum's elliptic curve digital signature algorithm (ecdsa). *Medium*, 2023.

[48] Oded Regev. crystals/kyber. `https://github.com/pq-crystals/kyber`, 2023.

[49] E. Rescorla. The transport layer security (tls) protocol version 1.3. *Internet Engineering Task Force (IETF)*, 2018.

[50] Redazione RHC. Il q-day si avvicina. e' necessario introdurre una crittografia resistente ai quanti. *Red Hot Cyber*, 2024.

[51] Open Quantum Safe. Oqs. *https://openquantumsafe.org/*, 2023.

[52] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 1997.

[53] Joseph H. Silverman. Ntru and lattice-based crypto: Past, present, and future. *The Mathematics of Post-Quantum Cryptography DIMACS Center, Rutgers University*, 2015.

[54] Kevin S.McCurley. The discrete logarithm problem. *American Mathematical Society*, 1990.

[55] Gorjan Alagic Daniel Apon David Cooper Quynh Dang Thinh Dang John Kelsey Jacob Lichtinger Yi-Kai Liu Carl Miller Dustin Moody Rene Peralta Ray Perlner Angela Robinson Daniel Smith-Tone. Status report on the third round of the nist post-quantum cryptography standardization process. *NIST*, 2022.

[56] Jacqueline Speiser. Implementing and comparing integer factorization algorithms. *Applied cryptography Group Stanford*, 2018.

[57] George Tasopoulos Jinhui Li Apostolos P. Fournaris Raymond K. Zhao Amin Sakzad Ron Steinfeld. Performance evaluation of post-quantum tls 1.3 on resource-constrained embedded systems. *Industrial Systems Institute/Research Center ATHENA*, 2021.

[58] SWilson4. liboqs. *github*, 2020.

[59] FALGUNI A. SUTHAR UNNATI P. PATEL, ASHA K. PATEL. The study of digital signature authentication process. *JOURNAL OF INFORMATION, KNOWLEDGE AND RESEARCH IN COMPUTER SCIENCE AND APPLICATIONS*, 2019.

[60] Nouri Alnahawi Johannes Müller Jan Oupický Alexander Wiesmaier. Sok: Post-quantum tls handshake. *University of Luxembourg Darmstadt University of Applied Sciences*, 2020.

[61] Tom Wiggers. Kem tls. *github*, 2023.

[62] www.geeksforgeeks.org. Data encryption standard (des). *geeksforgeeks*, 2023.

[63] Praveen K. Yenduri and Anna C. Gilbert. Continuous fast fourier sampling. *University of Michigan*, 2009.

[64] T. Ylonen. The secure shell (ssh) transport layer protocol. *Cisco Systems, Inc.*, 2006.

[65] Pierre-Alain Fouque Jeffrey Hoffstein Paul Kirchner Vadim Lyubashevsky Thomas Pornin Thomas Prest Thomas Ricosset Gregor Seiler William Whyte Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over ntru. *falcon@ens.fr*, 2020.