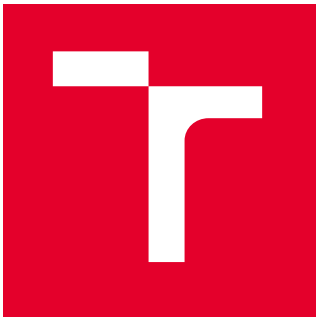
BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2023 Bc. Samuel Kopecký



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

MODULAR NETWORK COMMUNICATION USING POST-QUANTUM CRYPTOGRAPHY

MODULÁRNÍ KOMUNIKACE POSTAVENÁ NA POSTKVANTOVÉ KRYPTOGRAFII

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Samuel Kopecký**
AUTOR PRÁCE

SUPERVISOR — **Ing. David Smékal**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Information Security**

Department of Telecommunications

Student: Bc. Samuel Kopecký ***ID:*** 211799

Year of study: 2 ***Academic year:*** 2022/23

TITLE OF THESIS:

Modular network communication using post-quantum cryptography

INSTRUCTION:

The topic of the thesis is focused on the implementation of the library of post-quantum algorithms for key exchange, public key encryption and digital signature. The student implements at least one algorithm from the selected category.

In the diploma thesis, the student implements algorithm from each category and implements post-quantum client-server communication. The client part of the application will contain an API for downloading and uploading files and exchanging messages with other users.

The thesis will compare the speeds and performances with other implementations of post-quantum algorithms.

RECOMMENDED LITERATURE:

Podle pokynů vedoucího práce

Date of project specification: 6.2.2023 ***Deadline for submission:*** 19.5.2023

Supervisor: Ing. David Smékal

doc. Ing. Jan Hajný, Ph.D.
Chair of study program board

ABSTRACT

Current cryptography primitives, which are described at the begging of this thesis will be broken by future quantum computers. How they will be broken is described by this thesis along with a very basic description of quantum mechanics which are key to functional quantum computers. Available solutions like post-quantum cryptography are also introduced. More specifically code-based, hash-based and lattice-based cryptography. Lattice-based cryptography is described in most detail and specific NIST standardized algorithms are introduced – Kyber and Dilithium. Along with the theoretical description, an implementation is provided for both of the algorithms and a comparison to existing implementations in the programing language Go. Practical utilization of these algorithms is realized with a modular quantum-resistant communication application. It can send arbitrary data through a quantum-resistant secured channel and is well adjusted to the UNIX universal text interface. Notably it is able to exchange files between two users and also create a Terminal User Interface with which the users can communicate. The underlying protocol that is responsible for creating the secure channel is well defined in the latter chapters of this thesis. The modularity of the applications also allows users to remove or/and add any Key Exchange Mechanism or Digital signature which are responsible for the creation of the secure channel with very few code changes and good integration to the existing components of the application.

KEYWORDS

post-quantum cryptography, programming language Go, network communication, terminal user interface, lattice-based cryptography

ABSTRAKT

Súčasné kryptografické primitíva, ktoré sú popísané na začiatku tejto práce budú prelomené budúcimi kvantovými počítačmi. Táto práca popisuje proces lámania súčasnej krypografie spolu so základným popisom kvantovej mechaniky, ktorá je kľúčom k funkčným kvantovým počítačom. Taktiež predstavuje dostupné riešenia, ako je postkvantová kryptografia. Konkrétnejšie je predstavená kryptografia založená na kódoch, hašoch mriežkach. Najpodrobnejšie je opísaná kryptografia založená na mriežkach a sú predstavené špecifické NIST štandardizované algoritmy – Kyber a Dilithium. Spolu s teoretickým popisom je poskytnutá implementácia pre obidve algoritmy a porovnanie s existujúcimi implementáciami v programovacom jazyku Go. Praktické využitie týchto algoritmov je realizované modulárnou kvantovo odolnou komunikačnou aplikáciou. Je schopná posielať ľubovoľné dáta cez kvantovo odolný zabezpečený kanál a je dobre prispôsobená univerzálnemu textovému rozhraniu UNIX systmémoch. Viac špecificky, aplikácia je schopná vymieňať súbory medzi dvoma používateľmi a tiež vytvárať terminálové používateľské rozhranie, s ktorým môžu používatelia komunikovať. Protokol, ktorý je zodpovedný za vytvorenie zabezpečeného kanála, je dobre definovaný v posledných kapitolách tejto práce. Modularita aplikácie tiež umožňuje používateľom odstrániť a/alebo pridať akýkoľvek mechanizmus výmeny kľúčov alebo digitálny podpis, ktoré sú zodpovedné za vytvorenie zabezpečeného kanála s veľmi malými zmenami kódu a dobrou integráciou do existujúcich komponentov aplikácie.

KĽÚČOVÉ SLOVÁ

post-kvantová kryptografia, programovací jazyk Go, sieťová komunikácia, terminálové uživaťelské rozhranie, kryptografia založená na mriežkach

KOPECKÝ, Samuel. *Modulární komunikace postavená na postkvantové kryptografii*. Brno: Brno University of Technology, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2023, 91 p. Master's Thesis. Advised by Ing. David Smékal

Author's Declaration

Author:	Bc. Samuel Kopecký
Author's ID:	211799
Paper type:	Master's Thesis
Academic year:	2022/23
Topic:	Modulární komunikace postavená na postkvantové kryptografii

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I would like to thank the supervisor of the thesis, Ing. David Smékal, for his professional guidance, consultations, patience, suggestions and ideas.

Contents

List of Figures

List of Tables

Listings

Introduction

Until now the ever-increasing amount of computer power available was met with increased key or parameter sizes for existing cryptographic algorithms. For example, a few years ago it was sufficient enough to use RSA (Rivest Shamir Adleman) with 2048 bits of security, now 3072 bits of security are needed. The development of quantum computers is the next big step in technology evolution and brings many new possibilities for improvement but also many dangers to modern cryptography algorithms. For example, Shor's algorithm is capable of breaking modern asymmetric cryptography, which includes popular algorithms like RSA, ECDH (Elitic Curve Diffie-Hellman), and ECDSA(Digital Signature Algorithm).

A new approach to competing with the increasing computational power and new technologies had to be introduced. To combat the problem of possible quantum supremacy happening a new area of research has been created called post-quantum cryptography. It consists of cryptographic algorithms that are resistant to attacks using quantum algorithms or classical algorithms. One such group of post-quantum algorithms is lattice-based algorithms. They are the most promising group of post-quantum algorithms for standardization by NIST (National Insititue of Standards and Technology). NIST has so far led three rounds of the standardization process. During the writing of this thesis, the third round of NIST standardization ended and some of the winning algorithms are Kyber and Dilithium. Kyber is a KEM (Key Encapsulating Mechanism) and Dilithium servers as a digital signature algorithm. These algorithms are implemented in this thesis. There is also a 4th round where algorithms from the families of hash-based cryptography and code-based cryptography are competing.

The implementation language used for Kyber and Dilithium is Go. It was chosen because it creates a good balance between performance and simplicity. The performance is owned by the fact that it is a compiled language like C and shares many of its features. However, it also frees the programmer of many difficult and error-prone properties like memory management. In Go, it's solved using a garbage collector. To introduce the basic idea of these implemented algorithms, simplified block diagrams explaining the processes of Kyber and Dilithium are located in Appendix A.

Since these post-quantum algorithms have just been standardized recently, there aren't that many useful applications and programs that utilize them. A good first step would be to create a simple chat application or file-sharing application to showcase the security of these algorithms. One such application is introduced in this thesis.

1 Current state of cryptography

Cryptography is an essential part of Internet communication. It makes sure an established connection has three required properties [1]

- **confidentiality** – data can't be read by 3rd parties,
- **integrity** – data can't be edited by 3rd parties,
- **authenticity** – communicating parties can't be impersonated.

Many cryptographic primitives, algorithms and specifications exist in cryptography to ensure the aforementioned properties. The most commonly used protocol that utilizes these algorithms and specifications is TLS (Transport Layer Security).

Building blocks for cryptographic algorithms are cryptographic primitives. These are mathematical problems that can be solved in polynomial time ($O(n^x)$) with the knowledge of some secret. Without the knowledge of this secret, the problem can only be solved in exponential time ($O(x^n)$). This means if a new algorithm is found that can solve the problem without the knowledge of the secret in polynomial time, the underlying cryptographic primitive is broken and can no longer be safely used in any cryptographic algorithms or specifications.[2]

Cryptography can be split into symmetric cryptography and asymmetric cryptography. These groups and their underlying cryptographic primitives will be described in more detail in the following sections (Sections 1.1 and 1.3).

1.1 Symmetric cryptography

Symmetric cryptography is used for maintaining the confidentiality of data that is being transferred over a communication medium. The general idea of symmetric ciphers is that they are fast ciphers (compared to the asymmetric ones) that only use one secret (the secret key) to encrypt data. This key needs to be either pre-shared before the communication starts or a KEP (Key Exchange Protocol) has to be used (see Section 1.4). [3]

How symmetric ciphers work is illustrated in Figure 1.1. In a situation where Alice wants to send Bob a document (plaintext), Alice first needs to encrypt the document with the shared secret key. She then sends Bob the encrypted document (ciphertext) and Bob can decrypt it again with a shared key. Symmetric ciphers can be split into block and stream ciphers.

1.1.1 Block ciphers

Block ciphers operate on blocks of data and use padding to handle situations when a message can't be perfectly split into blocks. The same key is used for each block.

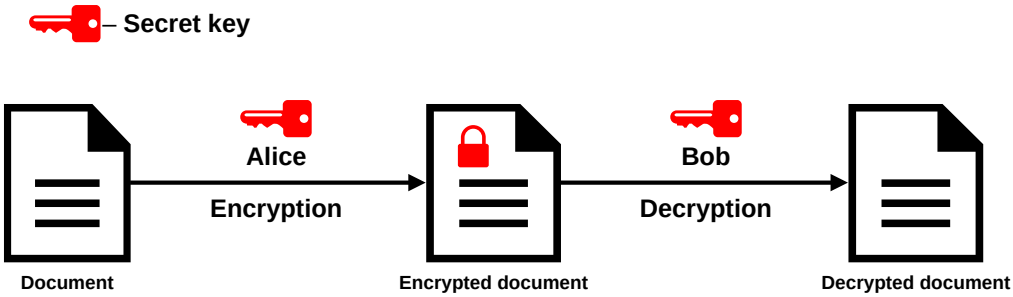


Fig. 1.1: Simplified symmetric cipher

Symmetric block ciphers can also use different modes of operation to add additional context to individual blocks from previous blocks. This process is important for the security of symmetric block ciphers because a block cipher without any mode of operation or an ECB (Electronic Code Book) mode generates the same output from the same input. This means an attacker could delete or add any block in an encrypted message without the receiver's knowledge. Some examples of a secure mode of operations for block ciphers are [4]

- **OFB** (Output Feedback),
- **CFB** (Cipher Feedback),
- **GCM** (Galois/Counter Mode).

Block ciphers are based on a substitution-permutation network (see Figure 1.2), which consists of two layers, a substitution layer and a permutation layer as the name implies. The substitution layer introduces confusion to the data. Confusion creates a correlation between the key and the ciphertext, where one changed bit in the key will generate a change for many bits in the ciphertext. In practice, a substitution layer just substitutes one byte with the help of a substitution table. This table is predefined and used for every operation. On the other hand, the permutation layer introduces diffusion, which means that a changed bit in the plaintext will dissipate into more changed bits in the ciphertext. In other words, it functions by scrambling the order of bytes randomly. An example can be seen in Figure 1.2 of a permutation layer. [4][5]

Of course, in practice, a cipher needs a lot more than just a simple substitution-permutation network. Good examples of block ciphers that use this principle are AES (Advanced Encryption Standard) and DES (Data Encryption Standard). DES is no longer deemed secure and should not be used [6]. AES on the other hand is still considered secure even to attacks from quantum computers if longer keys are used [7].

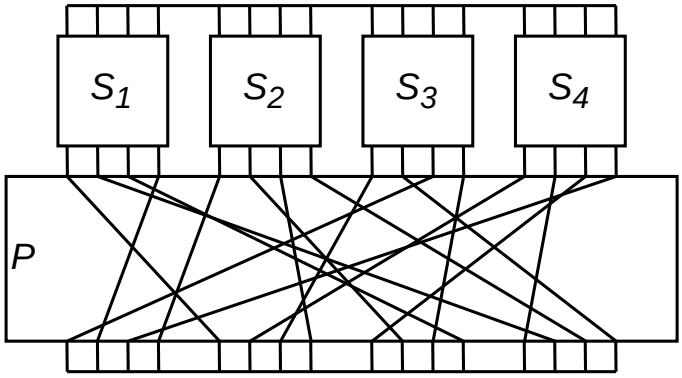


Fig. 1.2: Substitution-permutation network

1.1.2 Stream ciphers

Unlike block ciphers, stream ciphers encrypt one bit at a time instead of blocks. The main principle behind stream ciphers is the bit operation XOR and a PRNG (Pseudo Random Number Generator). The key is randomly generated by the PRNG function. Then the message is XORed with the generated key. The XOR operation can also be rewritten as mod 2 and thus the encryption process can be described in Equation 1.1

$$c = E(p) \equiv p + k \bmod 2 \tag{1.1}$$

and the decryption process in Equation 1.2

$$p = D(c) \equiv c + k \bmod 2 \tag{1.2}$$

for c as the ciphertext, p as the plaintext, k as the secret key E and D as the encryption and decryption functions respectively [4].

Examples of stream ciphers include RC4, Salsa20 or ChaCha20. It is no longer recommended to use the RC4 cipher. Salsa20 is a newer stream cipher and is considered to be resident even against quantum computers. [1][3]

1.2 Hash functions

Hash functions work by digesting a message of arbitrary size into a fixed-sized output or a variable-sized output (SHAKE family hash function) called the hash value. The digest process can also be described as a transformation of bits into another set of bits

$$H(k, n) : \{0, 1\}^k \rightarrow \{0, 1\}^d, \tag{1.3}$$

where k stands for the size of the input message and d stands for the output size.

For a hash function to be secure it also must possess these three properties [4]:

- **preimage resistance** – it is computationally infeasible to find the input of an already generated hash value,
- **second preimage resistance** – for a given hash value, it is computationally infeasible to generate two inputs that map to the same hash value,
- **collision resistance** – there mustn't exist two different inputs that generate the same hash value.

How the digest process works internally depends on the specific hash function being used, it doesn't have a single definition. For example, a hash function can be based on a Merkle-Damagård construction. This construction and many more use compression functions, which take in the input of some size and reduce it into an output of a smaller size. In the Merkle-Damgård construction, the message is firstly split into blocks. With the help of a compression function, the blocks are then consumed one by one. The output of one compressed block is then fed back to the input of another round of compression until all the blocks are consumed [2].

Other types of constructions are also used such as hash functions based on the KECCEK construction also called the sponge construction. The main idea behind the sponge construction is that after each round of compression, several bits are firstly absorbed by the compression function and then some bits are taken out of each compression iteration. These bits then make up the final hash value. How many bits are absorbed or taken out is dictated by the hash function parameters. [8]

Examples of specific hash functions are listed in Table 1.1. Since SHAKE can generate any sized output, its hash value size is dictated by the parameter d.

Tab. 1.1: Modern hash functions [8][9]

Algorithm	Underlying construction	Hash value size (bit)
SHA-256	Merkle–Damgård	256
SHA-512	Merkle–Damgård	512
SHA3-256	KECCAK	256
SHA3-512	KECCAK	512
SHAKE128	KECCAK	d
SHAKE256	KECCAK	d

Hash functions are used in many areas of cryptography. As an example, they are used in digital signature schemes (Section 1.3), message authentication codes (MAC), pseudo-random number generators and even public-key quantum-resistant cryptography [7].

1.3 Asymmetric cryptography

The other important type of cryptography is asymmetric cryptography also called public key cryptography. Compared to symmetric (see Section 1.1), asymmetric algorithms are most often slower, require bigger-sized keys and use two keys instead of one key. One of the keys that can be shared is the public key, the second key that has to be kept secret is called the private key. Depending on the use of these keys, asymmetric cryptography can be used in two ways – as an encryption cipher or as a digital signature scheme.

The principle of an encryption algorithm can be seen in Figure 1.3. Alice can encrypt a document using Bob's public key since the public key is shared with everyone and because Bob wants anyone to be able to send him encrypted messages. After receiving the encrypted document Bob can decrypt it with his private key since he is the only one that owns it. [2]

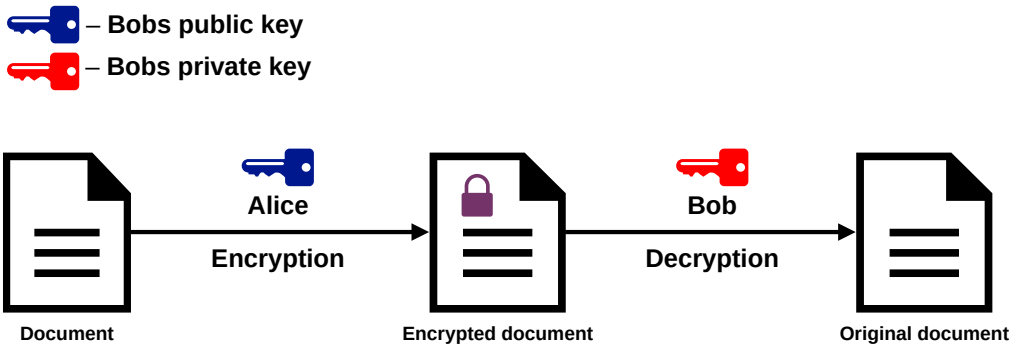


Fig. 1.3: Symplified asymmetric encryption cipher

Digital signature schemes serve as a tool to verify the origin of data. The following process is illustrated by Figure 1.4. If Bob wants anyone who receives his document to be able to verify that he was the one who created it, he signs the document with his private key. Everyone else including Alice can check whether the document came from Bob by verifying the signature with his public key. If the verification succeeds the verifier can be sure that Bob generated the signature because he is the only that posses the private key that generated the signature. [4]

In practice, Bob would be signing a hash of the document instead of the document itself, and would also send an unsigned document. Alice would then be comparing a hash of the document with the verified signature. This is because as mentioned before asymmetric algorithms are slow relative to symmetric algorithms and signing all of the data is unnecessary when signing the hash of some data servers the same purpose. Hash functions are described in the previous Section 1.2 of this chapter.

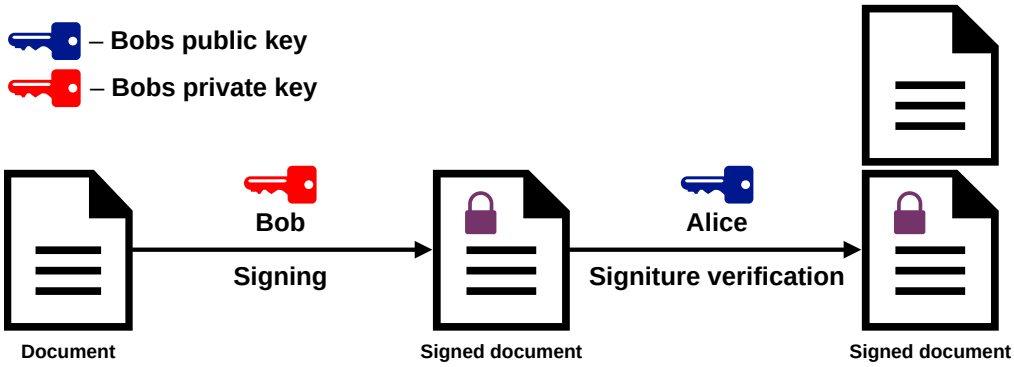


Fig. 1.4: Simplified digital signature scheme

1.3.1 Underlying principles

One of the underlying principles used in asymmetric cryptography is the integer factorization problem (IFP). This problem utilizes the idea that factorizing a big integer n composing of two prime numbers (more than 3072 bits) is impossible to compute on today's computers in polynomial time [4]. But producing n from two prime numbers p and q is trivial and fast

$$n = p * q. \tag{1.4}$$

where p and q are the unique prime numbers. IFP together with modular arithmetic create the RSA (Rivest Shamir Adleman) cipher that is one of the most used ciphers used today for creating digital signatures. The private and public keys are derived from the integer n.

The other principle that is used often in today's asymmetric cryptography is the discrete logarithm problem (DLP). It heavily relies on the use of modular arithmetic and cyclic groups in which there are a finite amount of integer values. This is possible because it uses the modulo operation together with other operations to stay inside this cyclic group. In this group, it is very easy (in polynomial time) to compute β with

$$\alpha^x \equiv \beta \bmod p \tag{1.5}$$

while knowing the values for x and α, but very hard (in exponential time on present-day computers) to compute x using this formula

$$x \equiv \log_\alpha \beta \bmod p \tag{1.6}$$

with the knowledge of only α and β, where p is a prime number with a bit size of at least 3072 [4]. DSA (Digital Signature Algorithm) utilizes this problem for creating digital signatures. An alternative algorithm exists that uses elliptic curves instead

of cyclic groups called ECDSA (Elitpic curve Digital Signature Algorithm). This is because the DLP equivalent in elliptic curves is more secure while using the same size for parameters such as x which is the private key [3]. This property allows the use of smaller keys while staying on the same level of security.

1.4 Key exchange protocols

As mentioned in Section 1.1, secret keys first need to be shared between the communicating entities before any encryption can begin. That is where a KEP (Key Exchange Protocol) is utilized. A subcategory of a KEP is a KEM (Key Encapsulating Mechanism). As they are a subset of asymmetric cryptography, many algorithms or ciphers used for asymmetric encryption can be converted to a KEM, for example, RSA [3]. How the key exchange works is illustrated by Figure 1.3, but instead of Alice encrypting and sending documents, she sends Bob a randomly generated encrypted key.

Another alternative of a KEP utilizes a dedicated key exchange method, such as the Diffie-Hellman protocol. It also works on the principle of having a public, private key pair like RSA, but each entity exchanges its public key with the other entity and then they calculate the shared secret key from the knowledge of their private key and the opposite entity's public key. Instead of relying on IFP, it relies on the DLP (see Subsection 1.3.1). This brings an advantage because the DH method can be then upgraded to Elitic Curve Diffie-Hellman (ECDH), which is a faster method for exchanging keys than plain DH [3].

2 Quantum supremacy

Modern cryptography described in Chapter 1 was designed with the assumption that the adversary would only have access to a classical computer. It turns out many of the algorithms and schemes used in modern cryptography are extremely vulnerable to quantum computers given the quantum computer has enough computational power [10]. Reaching a threshold of a powerful enough quantum computer is also called quantum supremacy. The following sections will explain what are quantum computers, how can they break classical cryptography and exactly which parts are vulnerable to quantum computers.

2.1 Quantum data representation

Quantum computers as the name implies, are based on the special properties of quantum mechanics. One of these many properties that quantum computers work with is the superposition of states. At very small sizes (sizes of individual particles) objects can be in such a state. Unlike ordinary objects of ordinary sizes, they can exist in more than one location at the same time. This phenomenon only occurs if the object is not being seen (is not being measured). However, this means whenever an object is measured in such a state the position of the object collapses into a single point in space. [11]

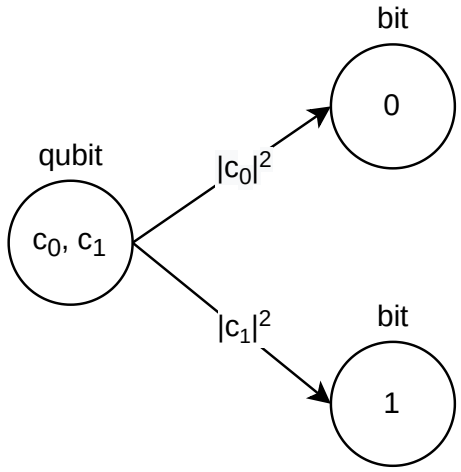


Fig. 2.1: Representation of a qubit

This unique property is what allows data to be represented in a quantum computer. In a classical computer, data is represented using bits. These only have 2 distinct values 0 or 1. Quantum computers don't work with bits but quantum bits or qubits in short. A qubit is represented by two pairs of complex numbers c_0 and c_1.

Complex numbers can be converted into real numbers p_0 and p_1 using Equation 2.1

$$\begin{aligned} p_0 &= |c_0|^2, \\ p_1 &= |c_1|^2, \end{aligned} \tag{2.1}$$

in this form they represent the probability of a qubit collapsing (after a measurement) into discrete values 0 or 1 and becoming a classical bit [11]. This concept is also illustrated in Figure 2.1 where the pointing arrows illustrate the qubit being measured.

By using complex numbers to represent qubits, they can be represented using the bra-ket notation

$$|\psi\rangle = c_0|0\rangle + c_1|1\rangle \tag{2.2}$$

where ψ represents the particle in a superposition of all possible states. A quantum computer can hold more than one qubit in a state of superposition. Unlike classical computers which always have one state, quantum computers can use the property of superposition and be in many states at the same time. This means it can evaluate a function for many values at the same time, which leads to great parallelism of quantum algorithms. However, a quantum algorithm doesn't work like a classical algorithm. It starts with a single position for all the qubits in the input. During the algorithm, the qubits are manipulated in their superposition state. When the algorithm finishes the state is then measured. At no point during the algorithm, the state can be measured, because then the superposition would be lost due to the qubits collapsing into a single state. [11]

2.2 Shor's algorithm

The biggest threat to modern cryptography is Shor's algorithm. It can be used for factoring prime numbers (N) in time complexity of $O(n^2 \log n \log \log n)$ where n is the number of bits required to represent N [11]. One of the fundamental problems used in modern cryptography is the IFP (see Subsection 1.3.1) used in RSA, which can be broken by Shor's algorithm in polynomial time. Shor's algorithm can be split into two parts. The first part can easily be computed on a classical computer, and the second part can also theoretically be done on a classical computer but it would take much longer than on a quantum computer.

The first part of Shor's algorithms is as follows. Generate a random number a in the range of $a \in \{2, \ldots, N-1\}$ which is co-prime to N, or

$$\mathrm{GCD}(a, N) = 1, \tag{2.3}$$

fortunately, we can use Euclid's algorithm to compute the GCD (Greatest Common Divisor) very fast even on a classical computer. From there the order of a has to be

found. The order is the smallest number such that

$$a^r \equiv 1 \bmod N. \tag{2.4}$$

Finding the order of a is computationally infeasible in polynomial time using a classical computer, that's why a quantum computer is needed to find r and will be explained later. If r is odd it is discarded and a new r is found by generating a new a. After the correct a is found, the Equation 2.4 can now be altered by subtracting 1 from both sides

$$a^r - 1 \equiv 0 \bmod N, \tag{2.5}$$

and now can be rewritten as

$$a^r - 1 = kN, \tag{2.6}$$

where k is some integer. With the help of $x^2 - y^2 = (x + y)(x - y)$ the previous equation can be written as

$$(\sqrt{a^r} + 1)(\sqrt{a^r} - 1) = kN, \tag{2.7}$$

or even a more readable version as

$$(a^{r/2} + 1)(a^{r/2} - 1) = kN. \tag{2.8}$$

Equation 2.8 can now be used to find at least one nontrivial factor of N by calculating

$$\begin{aligned} &\mathrm{GCD}((a^{r/2} + 1), N), \\ &\mathrm{GCD}((a^{r/2} - 1), N), \end{aligned} \tag{2.9}$$

and by dividing N with the first factor the second factor can be calculated and thus break any algorithm or cipher that depends on the IFP. After some modifications, it can also be used for breaking the DLP. [11][12]

The second part of the algorithm as mentioned is used for finding the order of a. An order of a number can also be represented as a period of the function $f_{a,N}(x)$

$$f_{a,N}(x) \equiv a^x \bmod N, \tag{2.10}$$

where its output values are repeated at regular intervals of size r [11]. The Table 2.1 shows an example for $N = 15$, $a = 2$ and $x \in \{0, 1, 2, 3, 4, 5\}$, where it is shown that the period (order) of a is $r = 4$. The repeating outputs can also be seen for $x \in \{5, 6\}$. As mentioned earlier, computing this on a classical computer for large N is infeasible but a quantum computer can evaluate a function for many values at the same time (see Section 2.1). This property is used in finding the order of a. It firstly calculates the repeating sequence of outputs for the function $f_{a,N}$ all at the same time. Using the QFT (Quantum Fourier Transform) the period is found which is the number r, then it can be used in the rest of the algorithm. [13]

Tab. 2.1: Example of a function period

x	0	1	2	3	**4**	5	6
$f_{a,N}(x)$	1	2	4	8	**1**	2	4

2.3 Grover's algorithm

Symmetric cryptography and hash functions can also be broken by another algorithm named Grover's algorithm. It is categorized as a search algorithm so instead of solving any mathematical problem, it just searches through all the possible options. Given a set of bits, $\{0,1\}^n$ where n is the size of the set a classical computer will search for a specific binary string of length n in $O(2^n)$ time. Grover's algorithm can search for the same binary string in $O(2^{n/2})$ time. [11]

Symmetric cryptography keys are also binary strings created from a set of bits size n, where Grover's algorithm can be used to find a key by trying all possible values. Similarly, hash functions also output a binary string from a set of bits. Grover's algorithm can be used to try to generate all the possible hash values inside a quantum computer and when a match is found it can retrospectively find the output that generated the hash value [14]. Since Grover's algorithm is not as efficient as Shor's algorithm in finding solutions that break ciphers or algorithms, key/hash value sizes can be increased to prevent these kinds of attacks [7].

2.4 Threat to modern cryptography

The future impact of quantum computers on classical cryptography can be seen in Table 2.2. ECC (Elitic Cruve Cryptography) algorithms and RSA aren't safe from quantum computers with a sufficient amount of physical qubits using Shor's algorithm. Currently, it is estimated that the required amount of qubits for Shor's algorithm to be efficient enough is in the tens of millions [1][15]. IBM managed to create a 433 physical qubit quantum processor in 2022 so humanity is not yet at the point where everyday internet communication using public key cryptography can be broken using quantum computers [16]. However, the threat is still there since traffic encrypted today using modern cryptography can still be broken later using quantum computers.

IBM has projected in their new roadmap to a practical quantum computer, that by 2025 they expect to have working quantum computers that contain around 4 158 physical qubits [17]. If this grows exponentially, a replacement for the current public key algorithms needs to be found. Each of the new candidates will be discussed in detail in Chapter 3.

Symmetric cryptography and hash functions on the other hand are much more resistant to quantum computers. For the current ciphers and algorithm to be quantum-resistant only the symmetric key size and digest size for hash functions needs to increase. For example, in the case of AES-128, it is sufficient enough to switch to AES-256 where the performance hit is negligible [1].

Tab. 2.2: Impact of quantum computers on classical cryptography[1][7]

Algorithm	Type	Impact
AES-128	Symmetric	Larger key sizes needed
Salsa20	Symmetric	Larger key sizes needed
GMAC	MAC	No impact
Poly1305	MAC	No impact
SHA2-256	Hash function	Larger output needed
SHA3-256	Hash function	Larger output needed
RSA-3072	Public key	No longer secure
ECDH-256	Public key	No longer secure
ECDSA-256	Public key	No longer secure

3 Post-quantum cryptography

Quantum supremacy may not be happening right now but may happen in the future. New public key algorithms need to be standardized so they can be used as replacements for quantum vulnerable algorithms such as RSA and ECDH. NIST (National Insititue of Standards and Technology) has begun the first standardization process for post-quantum algorithms, which are algorithms that are resistant to the future threat of quantum computers [7]. The main candidates that will be described in individual sections are

- lattice-based,
- code-based,
- hash-based.

3.1 Lattice-based cryptography

Lattice-based cryptography is said to be the most promising replacement for public key cryptography since two of them have already been standardized by the NIST (see Section 3.4) [18]. A lattice can be described as an infinite set of points in an n dimensional space. The space generated by these points is a periodic structure, an example can be seen in Figure 3.1. A lattice – the points in it – is generated by n linearly independent vectors which can also be called a base for the lattice [19]. Linearly independent vectors have the special property of not being a combination of any other vectors from the set of all vectors. An example of these vectors is illustrated in Figure 3.1. Vectors that generate a lattice can also be written in mathematical notation as

$$\mathcal{L}(B) = (b_1, \ldots, b_n) \tag{3.1}$$

where $\mathcal{L}(B)$ denotes a lattice created by a basis B. The basis is created from vectors $(b_1, \ldots, b_n)$.

To use lattices for cryptographic constructions, a vector v in a lattice has to be defined with coordinates from the set of all integers $\mathbb{Z}$. If every coordinate is then reduced with the operation defined as

$$v \equiv v \bmod q \tag{3.2}$$

where q is also an integer from $\mathbb{Z}$, the lattice is then called a q-ary lattice [10].

A cryptographic construction in lattices additionally needs a mathematical problem to be defined that can easily be calculated given in input but difficult to invert and calculate back the input that was given, in other words, a one-way function

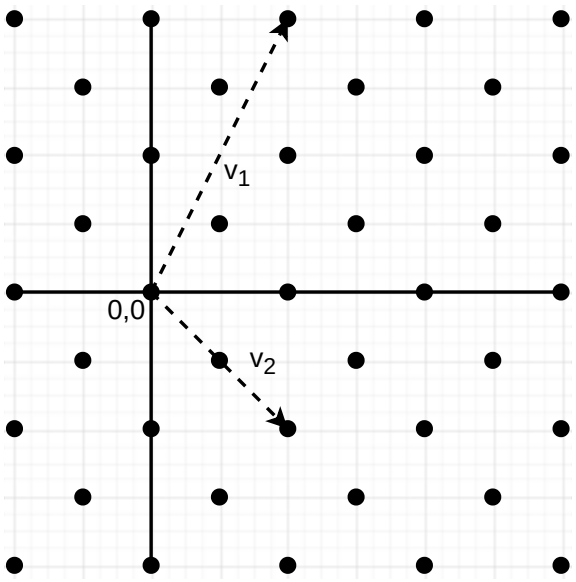


Fig. 3.1: 2-dimensional lattice

has to exist. One-way functions may also be described as a computational problem. In lattice-based cryptography there exist many computational problems, some of them are [10]

- **SVP** – Shortest Vector Problem,
- **CVP** – Closest Vector Problem,
- **LWE** – Learning With Errors.

How these computational problems are used and in which cryptographic algorithms or cryptosystems will be described in the following sections.

3.1.1 GGH public key cryptosystem

As mentioned in Figure 1.4 KEM (Key Encapsulating Mechanism) is one way of creating a KEP. In the case of lattice cryptography, an algorithm for creating a dedicated key exchange method like DH hasn't been found, so the only choice is to use a KEM. A KEM needs a public key encryption scheme to work, fortunately, many of them that use lattices have been discovered, so they can be used as key encapsulating mechanisms.

One of the first public key encryption schemes was the GGH cryptosystem which was named after its inventors Goldreich, Goldwasser and Halevi. Booth, the private and public keys are vector basis B and H respectively. A basis can also be written as a matrix where the columns of the matrix consist of the basis vectors. Additionally they form the same lattice $\mathcal{L}(B) = \mathcal{L}(H)$. The basis B is a good lattice and generates orthogonal or nearly orthogonal vectors. Basis H is called the bad basis and is

derived from basis B using a matrix T where

$$\begin{aligned} BT &= H, \\ HT^{-1} &= B. \end{aligned} \tag{3.3}$$

This transformation of B into H creates an orthogonality defect, which means the generated vectors by the basis are no longer orthogonal or close to orthogonal, this fact will be important later. The message to be encrypted is encoded into a vector v which is a lattice point in. Next a small noise vector e is chosen that is not a lattice point. Given these values the ciphertext c can be computed with $c = Hv+e$. The vector v or the plaintext can be extracted from c given $v = T\lfloor B^{-1}c \rceil$. The rounding operation is very important here since it removes the error that was added by vector e. [10][20]

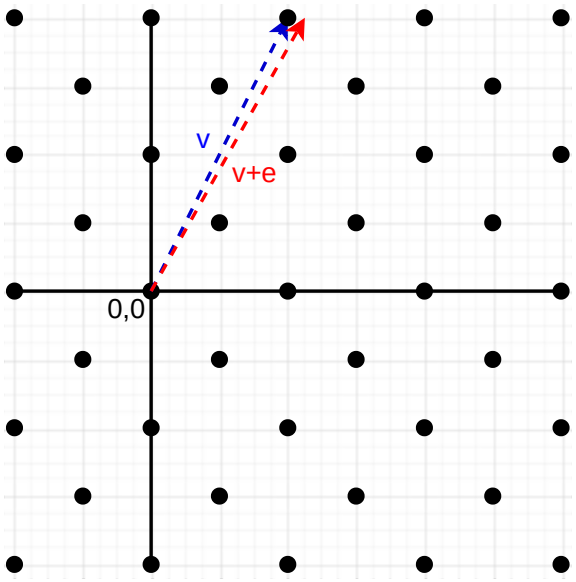


Fig. 3.2: Closest Vector Problem

Finding the original vector from the ciphertext is called the Closest Vector Problem (CVP) and is illustrated by Figure 3.2. The goal of this problem is to find the closest vector that is on a lattice point using a vector that isn't on a lattice point. The security of GGH relies on the fact that the CVP is easily computed while using the good basis B, but hard in the bad basis H. As mentioned earlier, a good basis is orthogonal and finding the closest vector is easily done using Babai's algorithm. However, this algorithm is inefficient in a basis that is not orthogonal, which in this case is the basis H. Based on this fact it can be assumed that only the owner of the good basis (private key) can decrypt a message. [20]

The only problem with GGH is that for it to be secure enough, it needs to have very big keys and as a result the computations are too slow. That is why this algorithm can't be used in practice [10].

3.1.2 NTRU and LWE public key cryptosystems

Another post-quantum KEM scheme is NTRU or N-th degree Truncated Polynomial Ring. It is one of the most efficient public key encryption schemes since instead of using a basis for its public key, it uses a polynomial consisting of p coefficients

$$h = h_0 + h_1 x + h_2 x^2 + \cdots + h_{p-1} x^{p-1}. \tag{3.4}$$

Like GGH this scheme is based on the CVP, so it uses a similar concept for the key exchange. Additionally, it generates the public key h where it is very efficient when preforming required operations. The use of polynomials makes NTRU much faster than the GGH cryptosystem and was considered heavily for post-quantum standardization by NIST [1].

LWE is not a cryptosystem by itself, but many cryptosystems are based on it. The problem is based on modular linear equations for example

$$3s_1 + 6s_2 + 7s_3 + 2s_4 \equiv 10 \bmod 11, \tag{3.5}$$
$$10s_1 + 8s_2 + 3s_3 + 5s_4 \equiv 1 \bmod 11, \tag{3.6}$$
$$5s_1 + s_2 + 7s_3 + 10s_4 \equiv 8 \bmod 11, \tag{3.7}$$
$$6s_1 + 8s_2 + 3s_3 + 4s_4 \equiv 7 \bmod 11, \tag{3.8}$$

where the goal is to find s_1, s_2, s_3, s_4. This is easily solvable even for big n amount of equations with the Gaussian elimination, but if an error is added to the right side of each equation (-1 or +1 in this case)

$$3s_1 + 6s_2 + 7s_3 + 2s_4 \equiv 9 \bmod 11, \tag{3.9}$$
$$10s_1 + 8s_2 + 3s_3 + 5s_4 \equiv 2 \bmod 11, \tag{3.10}$$
$$5s_1 + s_2 + 7s_3 + 10s_4 \equiv 9 \bmod 11, \tag{3.11}$$
$$6s_1 + 8s_2 + 3s_3 + 4s_4 \equiv 6 \bmod 11, \tag{3.12}$$

for big n it becomes a significantly harder problem. [21]

Unlike other mentioned lattice-based post-quantum cryptosystems, LWE-based cryptosystems are supported by a theoretical proof of security [10]. This makes them a very good candidate for standardization by NIST, more specifically the algorithm CRYSTALS-Kyber (see Section 3.4).

3.1.3 Digital signature schemes

As described in Figure 1.3, digital signatures are another branch of public key cryptography. Many of the same cryptosystems used for public key encryption can be converted into digital signature schemes, for example, booth GGH and NTRU.

However, the basic versions of these signature schemes have some security flaws that cause them to unusable in practice. [10]

The situation with an LWE-based cryptosystem is different. Digital signature schemes that use the LWE problem or a modified version of it called MLWE (Module Learning with Errors) are also very good candidates for standardization by NIST, for example, the digital signature scheme CRYSTALS-Dilithium [22]. More information on the topic of standardization can be found in Section 3.4.

3.2 Code-based cryptography

This family of post-quantum cryptography utilizes error correction codes. These are codes that can either detect or correct an error in some binary string by adding additional bits. However, they can correct/detect an error up to a threshold. If a big enough error is introduced, the code may no longer be able to detect or correct it.

The first ever code-based scheme was introduced by Robert J. McEliece in 1978 and so the scheme got the name from its inventor McEliece. The private key is defined as a random *Goppa code* which can correct errors in a coded sequence of bits. The public key is a matrix G and the plaintext m is a bit string. Additionally, another bit string is randomly created called e. The ciphertext c is then calculated with

$$c = mG + e. \tag{3.13}$$

Only the owner of the aforementioned *Goppa code* can extract m and e from c since the code was designed to efficiently correct errors added by the bit string e. [1]

Since this cryptosystem was introduced in 1978, it is well understood and has never been successfully broken. However for the system to be secure the private/public keys have to be relatively large compared to the keys of modern cryptography like ECDSA. On the other hand, they are very fast compared to the other algorithms submitted to the NIST standardization process. That is why 3 code-based algorithms are still being considered in the 4th round. [7]

3.3 Hash-based cryptography

Hash-based cryptography is mainly used for post-quantum digital signatures. Any hash function can be used inside a hash-based cipher as long as they are collision resistant. That means they don't rely on any hard mathematical problems, which makes their security requirement very low. Also because of this fact, every hash-based cipher can have many alternatives using many different hash functions. [10]

The first hash-based scheme was proposed by Leslie Lamport in 1975. If the output of the chosen hash function h is 256 bits, the private key x consists of 256 pairs of random bit strings, where each string is 256 bits long. The public key y is then generated by hashing every random bit string. At this point the public and private keys are

$$x = (x_{0,0}, x_{0,1}, x_{1,0}, x_{1,1}, \ldots, x_{256,0}, x_{256,1}), \tag{3.14}$$

$$y = (h(x_{0,0}), h(x_{0,1}), h(x_{1,0}), h(x_{1,1}), \ldots, h(x_{256,0}), h(x_{256,1})). \tag{3.15}$$

Given a message m which at first is hashed, the signature σ consists of either $x_{0,0}$ if the first bit of the message hash is 0 or $x_{0,1}$ if its 1. This repeats for every bit of the hashed message. The resulting signature for a given message is

$$h(m) = (01 \ldots 0)^{256}, \tag{3.16}$$

$$\sigma = (x_{0,0}, x_{1,1}, \ldots, x_{256,0})^{256}. \tag{3.17}$$

The verifier also hashes the message, generating the same hash. He then chooses hash values from the public key depending on the bit string of the message hash and creates y_p. Then he hashes each hash of the signature and generates $h(\sigma)$, which is

$$h(\sigma) = (h(x_{0,0}), h(x_{1,1}), \ldots, h(x_{256,0}))^{256}, \tag{3.18}$$

$$y_p = (h(x_{0,0}), h(x_{1,1}), \ldots, h(x_{256,0}))^{256}. \tag{3.19}$$

If $h(\sigma) = y_p$ then the signature is verified. However, the signer cannot reuse the same private key since it was already used. That is why this algorithm is called Lamport's one-time signature. [1]

The solution to one-time hash-based signatures was introduced by Ralph Merkle in 1979 and is called Merkle's tree signature scheme. Key generation starts with generating a binary tree which always has 2^n leaves, this is the master private key. Each leaf corresponds to a hash of Lamport's public key. Every two neighboring nodes are hashed together to create their parent node. In Figure 3.3 an example can be given for N_1 and N_2 where

$$N_3 = h(N_1, N_2). \tag{3.20}$$

After the tree is generated the master public key is the root of the tree, in this case, N_{15}. To sign a message, the signer chooses some random leaf node and signs a message with Lamport's one-time signature using the corresponding private key. In this case N_5. To verify this signature, at first, the one-time signature is verified using Lamport's public key. The signature is verified but the public key also has to be verified. That is why the signer also sent the least amount of hashes needed

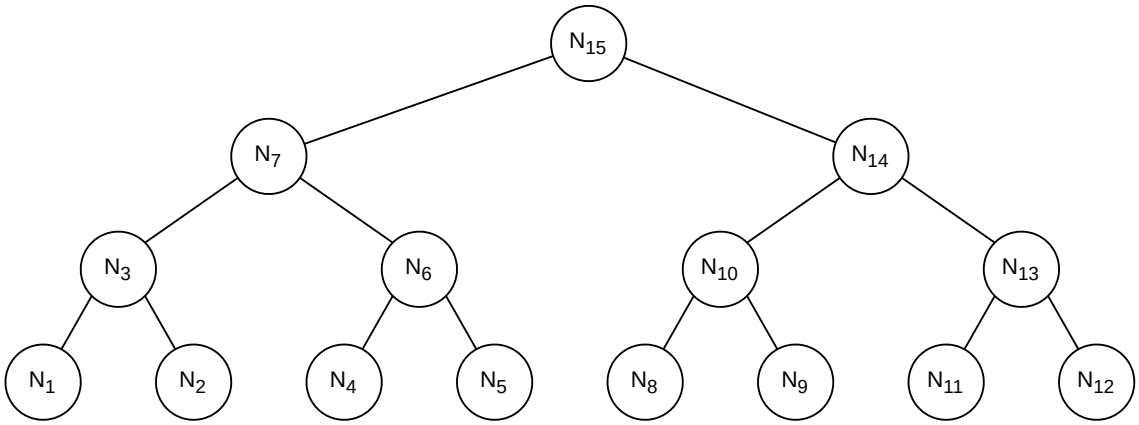


Fig. 3.3: Merkel tree

to compute the root. In this case he sent N_4, N_3, N_{14} as seen in Figure 3.4. To calculate N_{15}, the verifier has to calculate

$$N_6 = h(N_4, N_5) \tag{3.21}$$

$$N_7 = h(N_6, N_3) \tag{3.22}$$

$$N_{15} = h(N_{14}, N_7). \tag{3.23}$$

If the calculated N_{15} equals the master public key, the signature is verified. [10]

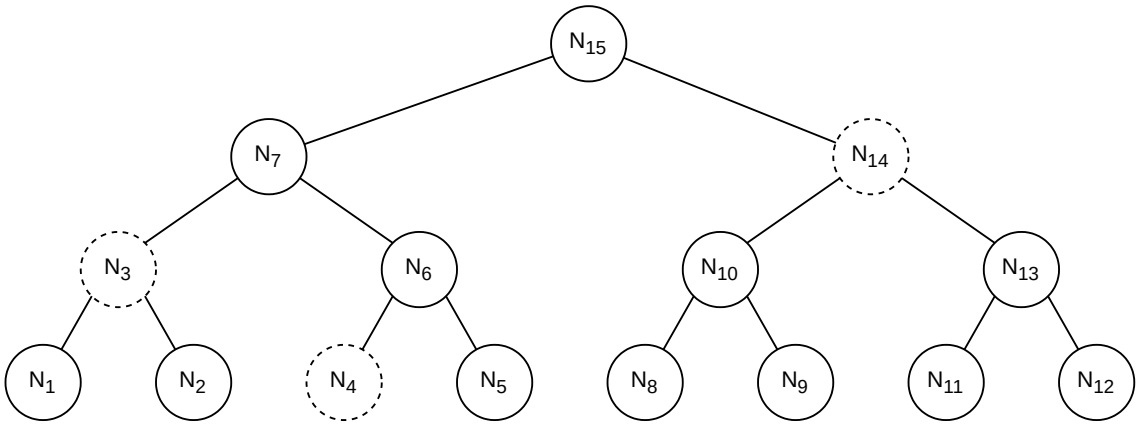


Fig. 3.4: Merkel tree – signature verification

As mentioned earlier, the security requirements for hash-based ciphers are very low and the principles that they are based on are very well understood. This makes them excellent candidates in the NIST standardization process. However, one flaw of these ciphers is that the signer has to keep a record of previously signed messages because they can produce only a limited amount of signatures. Although one signature has been standardized by NIST in the 3rd round and that is SPHINCS+, which is based on the aforementioned Merkle's tree signature.

3.4 NIST Standardization

NIST (National Insititue of Standards and Technology) started a standardization process in 2017 for the field of post-quantum public-key cryptography. The first call for submissions was initiated in December 2016, where 69 post-quantum algorithms were accepted into the first round of standardization. As of writing this thesis the latest round – the third round – ended on July 2022. The result was a standardization of 1 KEM protocol and 3 Digital signature protocols, for more details refer to Table 3.1. Additionally, four more post-quantum algorithms will be advancing to the fourth round of standardization, more information can be found in Table 3.2. [18]

Tab. 3.1: Standardized post-quantum algorithms [18]

Algorithm	Type	Based-on
CRYSTALS-Kyber	KEM	Lattice-based 3.1
CRYSTALS-Dilithium	Digital signature	Lattice-based 3.1
Falcon	Digital signature	Lattice-based 3.1
SPHINCS+	Digital signature	Hash-based 3.3

Tab. 3.2: Fourth round of NIST submissions [18]

Algorithm	Type	Based-on
BIKE	KEM	Code-based 3.2
Classic McEliece	KEM	Code-based 3.2
HQC	KEM	Code-based 3.2
SIKE	KEM	Isogeny-based

3.5 Disadvantages of post-quantum cryptography

Replacing modern cryptography like RSA with post-quantum algorithms is not as easy as it might seem. Quantum-resistant algorithms are certainly needed to prepare for the thread for quantum computers but one big disadvantage of these algorithms compared to modern cryptography is their computational requirements. Since they use more complicated structures and principles they also require more memory and processing power to compute. Some embedded devices might even take too long to compute some post-quantum algorithm to be useful or might just fail since they don't have enough memory. Another problem is the key sizes of these algorithms. They are a lot bigger as can be seen in Table 3.3 compared to modern cryptography. The chosen security parameters for the algorithms mentioned in the table below

correspond to the NIST security level of 3. Level 3 is defined as a security level that is only breakable by an attack that can break the AES algorithm with a key size of 192 bits or less [23]. Some embedded devices also have a very limited network bandwidth because they are battery-powered. These are the reasons why adapting post-quantum cryptography might not be as seamless as it might seem.

Tab. 3.3: Key size comparisons [24][25][26]

Name	Public key [B]	Private key[B]	Signature/Ciphertext [B]
RSA 7680	960	960	960
ECDH 384	48	48	48
Kyber 768	1 184	2 400	1 088
Dilithium 3	1 952	4 000	3 293

4 Network basics

To fully understand how any application that creates a communication channel between two entities works, it is important to first look at the concepts of network basics. The contents of this chapter will focus on topics such as the TCP/IP (Transmission Control Protocol/Internet Protocol) protocol suite, TCP, UDP (User Datagram Protocol) protocols and differences between client-server and peer-to-peer communications.

The TCP/IP protocol suite consists of 5 layers as can be seen in Table 4.1. Each layer is defined by one or more protocols. A protocol defines strict rules for what, how and when should an entity communicate [27]. Each layer in the TCP/IP suite contains at least one protocol, which again dictates how the communication should proceed on that layer. A protocol layer communicates vertically with other protocol layers using PDUs (Protocol Data Unit), each PDU is either encapsulated or de-encapsulated into another PDU, depending on the way the data is flowing through the TCP/IP layers. During this process, a new header is added or removed. A header contains important information for that specific layer. For example, the IP address is contained in the header for the network layer. The layers also communicate horizontally using either physical channels (physical and data link layers) or virtual channels (all other layers). Physical channels are created between a physical medium through which the bits travel, virtual channels on the other hand are created between applications on devices. Channels on each layer use a different identifier to differentiate between them. The aforementioned information for each layer can be found in Table 4.1.

Tab. 4.1: TCP/IP protocol suite [27]

(#) Layer	PDU	Identifier	Protocols
(5) Application	-	-	HTTP, FTP, SMTP, ...
(4) Transport	Segment/Datagram	Port	TCP, UDP, SCTP
(3) Network	Packet	IP address	IP, ...
(2) Data link	Frame	MAC address	Ethernet, ...
(1) Physical	Bit	-	-

4.1 TCP and UDP protocols

TCP and UDP are transport layer protocols in the TCP/IP protocol suite. The transport layer is responsible for creating connections between applications, where each application is identified with a port number which is stored in the transport

layer header. A port number can be any number in the range of 0-65535 [27]. For an application to be available it must listen on a port number so a client knows where to send his data. Similarly, if a client connects to an application he is also given a port number so that the server knows where to send his data. Booth TCP and UDP work with port numbers but an application can listen on the same port for both TCP and UDP protocols at the same time.

The UDP protocol was designed to be fast and unreliable [27]. It possesses these properties because it is a connection-less protocol. That means that there is no guarantee that the data that is being transferred will arrive as intended and without errors. It also means that the data can be sent faster and has less overhead communication compared to TCP. This model fits very well while sending very small amounts of data very quickly, like in the case of DNS translation. Before any data can be transferred, it has to be split into datagrams of smaller sizes. These are then sent one by one to the targeted entity.

The TCP protocol on the other hand is connection-oriented. Before data can be transferred between two entities first a connection has to be established using the three-way handshake (Figure 4.1) [27]. It works by setting bit flags in the TCP header, in this case, the SYN and ACK flags. After a connection has been established, the data transfer can begin. Unlike UDP, TCP is also numbering its segments which means it can detect if a segment was lost while being transferred, and then it can try to transfer it again. Another feature of TCP is flow control, which can be used for controlling how much data the communication entities can exchange at one time [27]. All of these features bring a much bigger overhead to each segment since more information needs to be tracked. This takes a toll on how fast segments can be transferred and also increases the size of the segments, which results in a slower but more reliable protocol than UDP. An example of good usage for the TCP is the HTTP protocol where a website needs to be transferred exactly as intended without any errors.

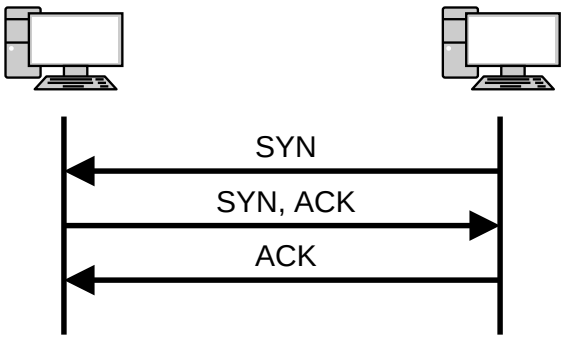


Fig. 4.1: Three-way handshake

4.2 Communication paradigms

Two of the most used communication paradigms to provide services to users are client-server and peer-to-peer, where the former is more commonly used in the Internet [27]. As the name implies a service is hosted on a server, and a client or more clients can connect to this server to consume the hosted service. Most of the time the server is a more powerful computer system so that it can handle more requests at the same time. Services may consist of providing some content to one user, for example, a simple website that provides HTML content. It can also provide a connection between two users so that they can communicate. For this, to work the server has to create multiple connections, one with each of the users. If the communication is encrypted, the server has to first decrypt an encrypted message, read it, encrypt it again and send it to the second user. The content of the exchanged messages was also seen by the server, which implies the users have to trust the server to not store or log their exchanged messages anywhere. However, if the client-server paradigm is enhanced with end-to-end encryption, the server doesn't have to decrypt/encrypt anything and just forwards the messages. For more information on end-to-end encryption see Section 4.3.

The second communication paradigm, peer-to-peer relies on the fact that if two entities want to communicate they will create a connection between them only, which eliminates the problem with the client-server paradigm of connecting two users. Each entity consists of a server and a client, since both of them need to be able to listen for incoming connections and also accept connections from other listening entities. This makes it also a derivative of the client-server paradigm just without the middle entity. It also makes it easier to implement end-to-end encryption.

The issue with using peer-to-peer for communicating with users is that each user needs to have an open port. Since most users on the internet are behind a NAT (Network Address Translation), it is not always easily solvable. This issue is mitigated by using the client-server paradigm since the clients can initialize the connections with the server which then transfers the messages between the initialized connections. However, as mentioned before this leaves the messages open and readable by the server if end-to-end encryption is not used. A comprise has to be made between the ease of use and the amount of trust one is willing to give.

4.3 End-to-End Encryption

E2EE (End-to-End Encryption) is a concept that allows data to be transmitted from one end user to another end user without being revealed or being tampered with along the way. It is mostly used in messaging apps. It is relatively simple to

implement using the peer-to-peer parading. Before communication starts, each of the end users converges on some private key using some key exchange protocol. One user uses this key to encrypt the data and the other user can then decrypt it. Of course in a real-life scenario where third parties try to attack this communication a lot of other things have to be considered like the integrity of the data and authenticity of the users.

However, in the client-server paradigm, it's a bit trickier to implement E2EE, since there is always some other entity between two end users. If the middle entity is a malicious one, it could easily use a man-in-the-middle attack on the key exchange protocol. Despite this, some protocols provide E2EE that ensures no third entity can utilize such an attack. One such example is the Signal protocol which uses the Double Ratchet Algorithm [28].

5 Application introduction

One of the goals of this thesis is to create an application capable of exchanging messages or files between two users. Secured only using quantum-resistant cryptography while relying only on the console environment for controlling the application. To exchange messages the application uses the peer-to-peer paradigm. However, there is still one peer called the user and one peer called the server to differentiate between the two peers. That is why in some places like the implementation code, client and server are used instead of peers. It is also important to mention that the application uses TCP as its transport layer protocol. It is implemented in the Go programming language. As the public key algorithms, CRYSTALS-Kyber and CRYSTALS-Dilithium are chosen. AES is chosen as the symmetric key encryption algorithm. Additional sections in this chapter explain why these choices were made and shortly introduce the chosen tools. The keyword modular in the thesis name signifies algorithm modularity in its implementation (explained further in Subsection 5.1.3).

5.1 Philosophy behind the application

The main philosophy of this application was inspired by a network protocol called Wireguard [29] and a book The Art of UNIX Programming [30]. It is meant to be a simple and small application for sending arbitrary amounts of data using only post-quantum cryptography. The four main ideas that are incorporated into the application are

- small traffic overhead,
- text-based interface,
- modularity,
- extensibility.

These ideas are introduced in the following sections.

5.1.1 Small traffic overhead

To achieve a small traffic overhead, this application uses the smallest header possible. The header only contains the data size and type. For a more detailed description of the header see Chapter 9. This makes the traffic overhead very low. Another mechanism that reduces traffic overhead is the number of initialization messages at the start of the connection. For example, the protocol TLSv1.2 uses handshake messages to negotiate on a cipher suite and then create a shared key using some key exchange protocol [3]. The protocol created in this application uses a pre-configured

cryptography suite and doesn't have to spend time negotiating it like TLS and only sends necessary messages to establish a secret key. This results in the protocol needing 1 round trip to initialize a connection. Again more information on the protocol flow can be found in Chapter 9.

However, one flaw is that this application uses TCP. To minimize traffic overhead UDP would be a better fit since it doesn't require connection initialization via the three-way handshake. It would also mean that things like flow control, error handling, synchronization and more would have to be solved in the application layer. TCP was chosen over UDP because it has been designed to do these things well and efficiently. As a result, the application needs 2 round trips to initialize a connection including the TCP handshake.

5.1.2 Text-based interface

From the start, this application was designed to be mostly used in Unix-like systems (but also supports other platforms like Windows). Most Unix programs and applications use a text-based interface, this way it's very easy to redirect an output of one program to the input of another program and vice versa. That is why this application is also designed to use the universal text interface so that it can easily interact with other UNIX programs. The data to send can be redirected via a pipe into the application. Received data can be saved to a file by redirecting its output. For more information about the capabilities of this application refer to Chapter 8.

5.1.3 Modularity

Modularity in the case of this application means that any other post-quantum algorithm can be easily plugged into the application without any massive code changes. This allows anyone to just add their preferred algorithm to the application in a relatively short time and without the knowledge of the whole code base. How algorithms can be added is detailed in Section 8.5.

5.1.4 Extensibility

This application is made to be extensible, which means it is very easy to use the underlying communication protocol for sending arbitrary amounts of data and extending it with additional functionality. For example, if another programmer wants to implement a new GUI (Graphical User interface), the application was programmed in a way where it's easy to use the underlying secure communication protocol. The programmer doesn't have to understand how the protocol works, he just needs to know how to use it. More specifically it uses a Go feature called channels for sending

and receiving data. When data is sent to the channel it is automatically encrypted and similarly, if some data is received from a channel it is already verified and decrypted.

5.2 The Go programming language

The first iteration of the Go programming language was created at Google. It is an open-source programming language and has many similarities with C. That means it is a compiled language and a statically typed language [31]. In a statically typed language, a variable has to have a type assigned to it before the compilation process. As can be seen in Figure 5.1 a compiler translates the source program into an executable, which can run multiple times without the need to compile again [32]. This makes Go faster than most of the interpreted languages like Python since an interpreter needs to translate the source code every time it has to run.

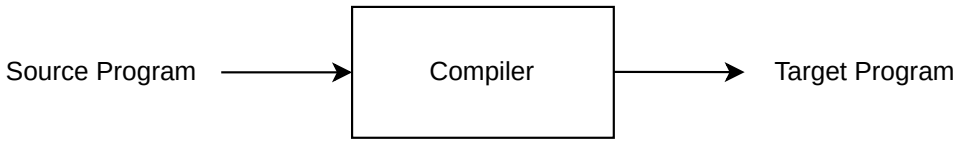


Fig. 5.1: Compiler

However unlike C, it has a garbage collector, which means it is capable of automatic memory management [31]. In C a programmer has to manage memory on his own, allocate and free it by using functions. Go and its garbage collector takes care of allocating and freeing memory which makes it a lot less error-prone when it comes to memory management. Together with good overall performance, Go was also designed to make high-performance network applications that's why it was chosen for this thesis.

5.3 Choice of cryptography algorithms

As mentioned in Section 3.4 a group of three lattice-based algorithms has been standardized by NIST, of which Kyber and Dilithium are a part of. That is the main reason why they are chosen as the public key algorithms for this thesis. Since the application is modular any security level of Kyber and Dilithium can be used but the default choice is Kyber1024 and Dilithium5. Other possible choices for these algorithms can be seen in Appendix C. Each of them has its respective Chapter (6 and 7) which describes some implementation aspects and also introduces some theoretical background that makes them secure.

Tab. 5.1: Choosen algorithms

Type	Algorithm
Digital signature	Dilithium5
Key exchange/Public key encryption	Kyber1024
Symmetric encryption	AES
Hash function	SHAKE-128/256, SHA3-512

As for symmetric cryptography, AES is chosen with a key size of 256 bits. AES is used in modern cryptography as the main symmetric cipher and is also usable in post-quantum cryptography but requires bigger-sized keys, due to the threat of Grover's algorithm. Used hash functions in this application are SHAKE-128/256 for the implementation of Kyber and Dilithium following the author's recommendations. SHA3-512 is used by the networking part of the application. A summary of the chosen cryptographic algorithms can be found in the Table 5.1.

6 CRYSTALS-Kyber

CRYSTALS (Cryptographic Suite for Algebraic Lattices) Kyber is a quantum resistant KEM standardized in the 3rd round of the NIST standardization process. It is based on a modified version of the LWE problem (Section 3.1.2) called MLWE (Module Learning with Errors). Lattices have a solid theoretical security foundation because they have been researched for a long time and are not a new invention. NIST has concluded in their report of their 3rd round of standardization that Kyber has sufficient security against quantum computer attacks. Even in the worst-case scenario where the development of quantum computers is underestimated. As to performance, it has been shown that Kyber is the fastest algorithm amongst the other lattice KEM NIST finalists when it comes to key generation, encapsulation and decapsulation in software and hardware. [22][18]

6.1 Implementing Kyber

The implementation of Kyber in this thesis is done using only the standard Go library except for one external library called crypto[1] which is required for the implementations of the hash functions SHAKE-128/256 (refer to Section 1.2 for hash functions). Figure A.1 illustrates a very simplified block diagram of how Kyber works. Individual blocks in this figure represent a mathematical structure or a variable in a program. This structure is almost always composed of polynomials which represent a ring and is described in Section 6.2. A collection of polynomials can also be called a vector of polynomials. The small letter or number at the start of an arrow coming from the structures denotes the size of the structure. In the case of a vector, it denotes the number of polynomials the vector contains. So for example the letter k denotes that a vector consists of k polynomials.

As to the process of how Kyber works, firstly the public and private keys need to be generated. A random message m is then encrypted by one communicating entity using the public key. The encrypted message is then decrypted by the other communicating entity and m becomes the shared key K. The following subsections will explain each sub-algorithm of the block diagram in more detail. The Go code in the practical part is also a great reference to understand how Kyber works. These functions are then wrapped by another set of three functions which are also key generation, encapsulation and decapsulation. This is done to provide additional security for the Kyber scheme.

[1] `https://pkg.go.dev/golang.org/x/crypto`

Tab. 6.1: Kyber security levels [25]

	n	k	q	η_1	η_2	(d_u, d_v)
Kyber512	256	2	3329	3	2	$(10, 4)$
Kyber768	256	3	3329	2	2	$(10, 4)$
Kyber1024	256	4	3329	2	2	$(11, 5)$

Kyber uses a set of parameters to define its security level, of which it has three as seen in Table 6.1. Kyber in this thesis is implemented for all the parameter levels. What individual parameters mean will be explained in further subchapters.

6.2 Theoretical background

Kyber uses a structure called rings, more specifically the ring R_q denoted as

$$\mathbb{Z}_q[X]/(X^n + 1). \tag{6.1}$$

A ring contains a polynomial of n elements, where the coefficients of this polynomial are integers reduced modulo q and the powers of the polynomial are reduced (X^n+1). The parameters n and q are defined in Table 6.1. An example of a polynomial with elements from the ring R_q is

$$t_1 = 1564 + 2189x + 258x^2 + \cdots + 655x^{n-2} + 2587x^{n-1}. \tag{6.2}$$

A vector of size k consists of k polynomials with coefficients from the ring R_q. The parameter k can be found in Table 6.1. The the polynomial t_1 from the previous example together with a new polynomial t_2

$$t_2 = 2408 + 1932x + 420x^2 + \cdots + 3256^{n-2} + 2399^{n-1}, \tag{6.3}$$

form a vector of polynomials $\mathrm{T} = (t_1, t_2)$. A matrix of size $k \times k$ consists of k^2 polynomials from the ring R_q aligned as a square 2-dimensional matrix. [25]

The addition of elements from a ring is just adding the individual polynomials and is relatively fast. Multiplication of vectors or matrices the usual way (multiplying each element by each element of the other polynomial) is computationally much more demanding with big n. In this case where $n = 256$ the number of computations would be $n^2 = 262144$. A more efficient way to calculate the multiple of two polynomials is using an NTT (Number Theoretic Transform) where the number of operations is only $n \log(n) = 1387$. This transformation is a more specific version of the FFT (Fast Furier Transform). However, before doing the NTT multiplication it is first required to transform the polynomial into NTT form. Do the calculation with

some other polynomial in NTT form and then do the inverse NTT transformation on the result. In this thesis structures that are converted to NTT are denoted with a hat, for example, $\hat{A}$. [33]

6.3 Encoding, Compression and randomness

To transfer polynomials over the network they need to be serialized into bytes. Kyber defines two functions for this purpose:

- `encode`(p, l) – convert a polynomial into $32 * l$ bytes,
- `decode`(B, l) – convert $32 * l$ bytes into a polynomial.

Since these functions only work on a polynomial, if they are to be applied to a vector of polynomials, every polynomial is processed separately.

Another functionality of Kyber is the compression of polynomials that are encoded. Because Kyber is based on LWE, the calculations don't have to be precise just close enough. This is why a compression mechanism that discards some low-order bits from encoded polynomials can be utilized. Two more functions are defined by Kyber for compressing and decompressing bytes

- `compress`(x, d) – compress a number into the range of $\{0, \ldots, 2^d - 1\}$,
- `decompress`(x, d) – decompress a number while loosing some low-order bits.

In order to apply this transformation to a polynomial vector each polynomial is transformed separately. The functions are used on each coefficient of the polynomial. The parameters d_u and d_v are used as the inputs for these functions.

Polynomials need to be randomly generated in Kyber. A CBD (Centrail Bionimal Distribution) function is defined which takes as an input in a byte array of some parameter η multiplied by 64. It then generates a fixed length polynomial of size n. This function is also used in generating random vectors where each polynomial is again generated separately. The input η value can either be η_1 or η_2 defined in Table 6.1.

6.4 Key generation

The key generation functions starts by generating random parameters (illustrated by Figure 6.1). A random seed ρ is used to generate the matrix $\hat{A}$. It is publicly known to everyone and needs to be shared. However, since the function that generates it is deterministic only ρ needs to be shared instead of the whole matrix. This mechanism saves a lot of network traffic because the matrix $\hat{A}$ would consume a lot more network traffic than just sending ρ. Two vectors s and e are generated from a different random seed. In this case, the seed is not shared since s and e need to

remain secret. After transforming the generated vectors into the NTT domain, $\hat{A}$ and $\hat{s}$ are multiplied. The vector $\hat{e}$ is then added to the result and creates the public key. The encoded vector s is then used as the private key.

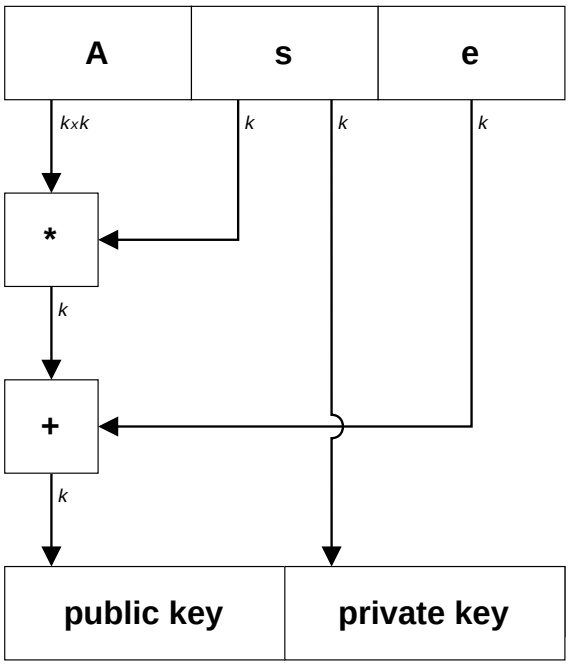


Fig. 6.1: Kyber key generation

6.5 Encapsulation

The encapsulation process relies on the encryption function (Figure 6.2) and will be explained further. Firstly the parameters have to be set up. The public key is decoded into $\hat{t}$. The matrix $\hat{A}$ is generated from ρ which is also a part of the public key. A random polynomial vector r is created and transformed into the NTT domain. Parameters e_1 and e_2 are also randomly generated where the first one is another polynomial vector and e_2 is just a single polynomial. The last required parameter is the randomly generated message m.

$\hat{A}$ and $\hat{r}$ are multiplied and e_1 is added to the result. Afterwards, it is transformed from the NTT domain since booth factors are in the NTT domain. The result of these operations is u which forms a part of the ciphertext. The message m is decoded to create a polynomial from it and decompressed. A polynomial is calculated using the factor of $\hat{t}$ and $\hat{r}$ and is again similarly transformed from the NTT domain. Then the polynomial e_2 is added to it together with the decoded message. To create the ciphertext booth u and v are compressed and encoded to get them ready for

network transfer. The parameters used in the compression and encoding processes are d_u and d_v reference in Table 6.1.

After the encapsulation process is done the entity that generated the random m uses it to generate a random K that will be used for some other purpose like a secret key for symmetric encryption.

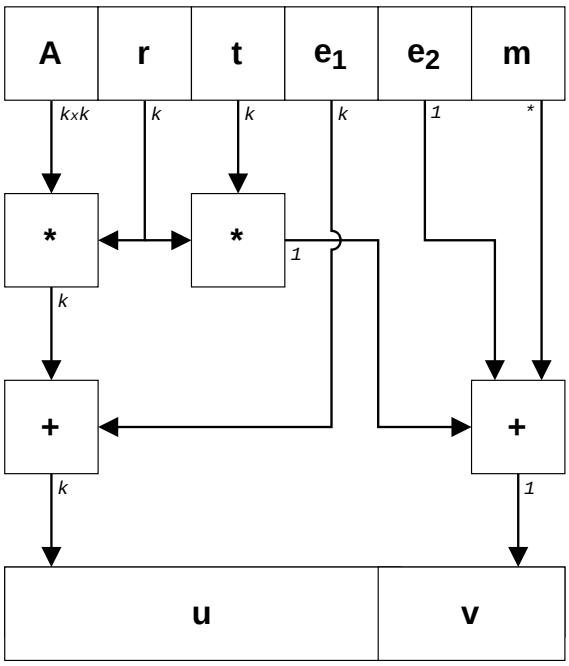


Fig. 6.2: Kyber encryption function

6.6 Decapsulation

Similarly, as with the encapsulation process, the decapsulation process requires the decryption function to be defined. The decryption process contains only a few calculations and is illustrated by a figure, specifically 6.3. It begins with decoding and decompressing the parameters u and v from the ciphertext. Additionally, u is transformed into the NTT domain. The private key is also decoded into the vector $\hat{s}$. The actual decryption begins by multiplying $\hat{s}$ and $\hat{u}$ and transforming the product from the NTT domain. It is then subtracted from v compressed and decoded to get the original message m. After the message is decrypted it can be used to generate the same key K that will be used further.

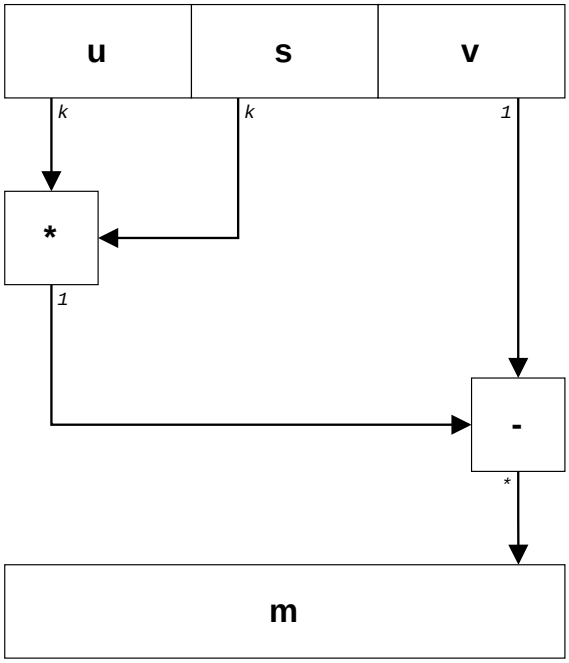


Fig. 6.3: Kyber decryption algorithm

7 CRYSTALS-Dilithium

Another algorithm from the group of lattice-based cryptography is the CRYSTALS-Dilithium signature scheme. It was also standardized during the 3rd round of the NIST standardization process on post-quantum cryptography. It is based on the Fiat-Shamir paradigm which means a prover can convince a verifier of the fact that they hold a private key without actually revealing it. Similarly, Kyber is also based on the MLWE problem. Dilithium also has a binding property that allows a signature to be linked with a unique public key and a message. When it comes to the security of Dilithium, it is proven that a signature is unforgeable by classical and quantum computers. NIST mentioned in their report on the 3rd round of standardization that Dilithium has a strong security basis and along with Falcon is one of the most efficient signature algorithms. [18]

7.1 Implementing Dilithium

As with Kyber, Dilithium is implemented using only the standard go libraries and one external library named crypto[1] that contains implementations for SHAKE-128 and SHAKE-256 hash functions. Dilithium can be implemented in two ways, the first one is by using a bigger public key. This implementation of Dilithium is also simpler overall. The other option is implementing a more complex algorithm that has a smaller public key by a factor of more than half. For this thesis, a more complex implementation was chosen. How this alternative differs from the simpler one and will be explained in Subsection 7.2.1. The algorithms as a whole are described in Figure A.2. Analogous to the Kyber algorithm figure, each square represents a mathematical structure or a program variable, where the structures mostly represent a vector of polynomials. The small letters at the beginning of the arrows denote the number of polynomials that the resulting structure consists of.

The process of signing in Dilithium follows a well-defined order as with many other digital signatures. Firstly the public/private keys are generated and the private key is used in the signing process. This is key is not shared and kept secret by the signer. The result of a signing process is a signature that can be verified by anyone who owns the related public key. Since the public key is shared, it is not kept secret by the signer. The following sections will explain all of these steps in more detail. For an even more detailed description of Dilithium, check the algorithm implementation in the practical part of this thesis.

[1] `https://pkg.go.dev/golang.org/x/crypto`

Tab. 7.1: Dilithium security levels [26]

	n	q	d	τ	γ_1	γ_2	(k, l)	η	β	ω
Dilithium 2	256	8380417	13	39	2^{17}	$(q-1)/88$	$(4, 4)$	2	78	80
Dilithium 3	256	8380417	13	49	2^{19}	$(q-1)/32$	$(6, 5)$	4	196	55
Dilithium 5	256	8380417	13	60	2^{19}	$(q-1)/32$	$(8, 7)$	2	120	75

Table 7.1 displays the individual parameters for each of the Dilithium parameter sets. The implementation in this thesis contains all Dilithium security modes. When a parameter is relevant to the process being explained it will be mentioned and explained in that scenario instead of all the parameters explained in this section.

7.2 Bit manipulation

Dilithium employs some helper functions which are used in both the simple and more complex versions of Dilithium. The first one is `Decompose` and can be well explained using an example

$$\texttt{Decompose}(5687946,\ 1735) = 3278 * 1735 + 616. \tag{7.1}$$

As can be see in Equation 7.1 the `Decompose` function splits a number into two smaller numbers $r_1 = 3278$ and $r_0 = 616$. The number r_1 is the closest multiple of the second input parameter $\alpha = 1735$ to the input number. The second returned number r_0 is what remains after the division of α. This function is wrapped by two additional functions `HighBits` and `LowBits`. `LowBits` returns only r_0 and `HighBits` returns only r_1. A similar function `Power2Round` does basically the same but instead of taking any α as the divisor it takes a parameter d which is then used for calculating a power of 2 that is used as the divisor. Booth function output the same number for parameters $d = 13$ and $\alpha = 8192$ as seen bellow

$$\texttt{Power2Round}(5687946,\ 13) = 694 * 8192 + 2698, \tag{7.2}$$

$$\texttt{Decompose}(5687946,\ 8192) = 694 * 8192 + 2698. \tag{7.3}$$

Functions `MakeHint` and `UseHint` make use of the aforementioned functions to create and consume hints. These functions are only used in the more complex implementation. Their role is to reduce the size of the public key without sacrificing security. Firstly the `MakeHint` function is used to check whether the addition of z and r would change its high bits. If it, would the returned value is true and a hint is made. If is small enough to not change the high bits a hint isn't made. This process can be seen in Listing 7.1 which describes its implementation.

Listing 7.1: `MakeHint` implementation

```
1 return highBits(r, alpha) != highBits(r+z, alpha)
```

Now that the hints are created, they can be consumed with the function `UseHint`. This function is capable of reconstructing only the high bits of $r+z$ using the created hints without knowing z. It works by adding or subtracting 1 from r_1 depending on the sign of r_0 if a hint for that number was made. If it was not made r_1 stays the same. This concept can also be described by Equation 7.4.

$$\texttt{UseHint}(\texttt{MakeHint}(z,\, r,\, \alpha),\, r,\, \alpha) = \texttt{HighBits}(r + z,\, \alpha). \tag{7.4}$$

7.2.1 Reducing the public key

As mentioned the functions described in Section 7.2 are used in the complex implementation of dilithium to reduce the size of the public while providing the same security. Some of the bits from the public key are transferred to the private key. This makes the private key bigger but also makes the public key smaller. However now when verifying a signature, the calculations are not precise enough when only using the cut of the public key. However, during the signing process, hints are made using the `MakeHints` function. This is possible because some bits of the public key are stored in the private key. This is what allows the verification calculation to be just precise enough to correctly decide whether the verification is correct. [26]

7.3 Theoretical basics and bit packing

Dilithium uses the same theoretical background as Kyber (see Section 6.2) which includes rings, NTT transformation, polynomials and even uses the same n as can be seen in Table 7.1. However, the parameter q is different.

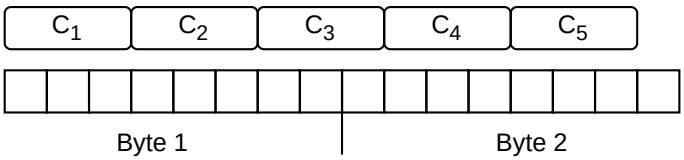


Fig. 7.1: Bit packing for vectors s_1 and s_2

Since Dilithium needs to transfer polynomials over the network a very efficient bit-packing method can be used. For example the polynomial vectors s_1 and s_2 consist of values that are from the interval $\{-2,\, -1\,, 0,\, 1\,, 2\}$ while using $\eta = 2$. This

means only 3 bits are required to pack a single coefficient into bit form (illustrated in Figure 7.1). However, the coefficients firstly need to be mapped into an interval $\{0, 1, 2, 3, 4\}$ while packing and moved back to $\{-2, -1, 0, 1, 2\}$ in the unpacking process. As a result one polynomial of the mentioned vectors only takes up 96 B and the whole polynomial vector only takes up 384 B. This is a big difference compared to simple packing where one byte contains one value. A very similar process is used for packing other polynomial vectors in Dilithium, the only difference being the size of the coefficient interval. Using a slightly different packing method for each kind of coefficient interval is what makes the bit packing/unpacking a very efficient method for encoding/decoding data that has to be sent over a network. This method is also used when a vector needs to be consumed by a hash function since it can only accept a byte array as its input.

7.4 Key generation

The dilithium key generation process starts by generating random seeds ρ and ρ'. ρ is used for generating the matrix $\hat{A}$ where its dimension are $k \times l$. For generating the error vectors s_1 and s_2, ρ' is used. The range of values in these vectors depends on the parameter η. Similarly, as with Kyber, only ρ is sent over the network since the function to generate $\hat{A}$ is deterministic. The product of $\hat{A}$ and s_1 to which s_2 is added is passed to the aforementioned function `Power2Round` together with d which is a parameter defined in Table 7.1. This function splits the results (t) into t_1 and t_0. This is the splitting that is talked out about in Subsection 7.2.1.

Variable t_1 is used as the public key together with the randomly generated K. The private key consists of a hash (in the Figure 7.2 this a block with H) of the public key tr, private parameters s_1, s_2, t_0 and the random seed ρ. Polynomial vectors are additionally packed into bytes for easy transfer over the network. This process is illustrated by Figure 7.2.

7.5 Signature creation

At the beginning of the Dilithium signing process, the private key has to be parsed into variables that it consists of. This is done by unpacking the bytes into useful data (see Section 7.3), more specifically the vectors s_1, s_2, tr and t_0. The vectors s_1, s_2 need to be converted to the NTT form but this can be precomputed ahead of time to increase the speed of singing. The parameter $\hat{A}$ is generated from the shared seed ρ. The message to be signed is hashed together with tr which is used for generating the vector y. However, for the sake of simplicity and clarity, this

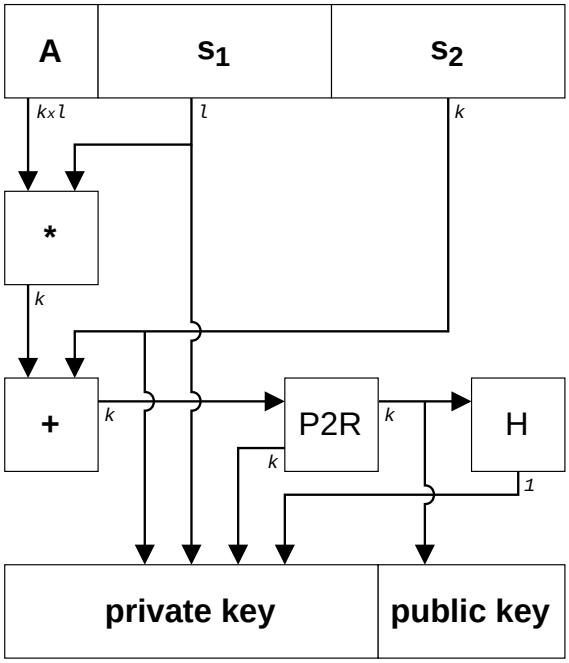


Fig. 7.2: Dilithium key generation

generation process is not described in Figure 7.3 and the parameter y is just shown as one of the inputs. Next the product of $\hat{A}$ and $\hat{y}$ which is denoted w is calculated. High bits of w (`hb(k)` in Figure 7.3) are hashed together with the hash used for generating y, to create the polynomial c. It is first used for multiplying vectors s_1 and s_2. After that is used as a part of the signature. The vector y is added to cs_1 (cs_1 is s_1 scaled by c) and creates another part of the signature, the vector z. The subtraction of the vectors w and cs_2 (vector r) is added together with the scaled vector ct_0. The result of this process is used as the second input for the function `MakeHint`. The first input is ct_0 but negated. `MakeHint` functions return the final value for the signature. These hints will then be used to calculate the missing part of the public key as described in 7.2.1.

However, before the signature creation is finalized, a few conditions have to be met. If these conditions are not met, most of the signing process is repeated. Some of the conditions mention parameters defined in Table 7.1. These conditions are

- the polynomial coefficients in z can't be bigger than $\gamma_1 - \beta$,
- the polynomial coefficients in r_0 (`LowBits` of r) can't be bigger then $\gamma_2 - \beta$,
- the polynomial coefficients in ct_0 can't be bigger then γ_2,
- number of created hints can't be more than ω.

7.6 Signature verification

The verification process for Dilithium starts by unpacking the required variables from the public key and the signature. $\hat{A}$ is generated the same way as in the signature generation algorithm, it is then multiplied by the vector z from the public key and called Az. The public parameter t_1 is at first scaled by 2^d where d is defined by Table 7.1. This is making up for the lost bits during the key generation. It is then scaled again by c which is parsed from the signature. The result of subtracting ct_1 from Az is used as the first input for the `UseHint` function. The second input parameter is h from the signature. Lastly, the hash of the signed message and t_1 is hashed together with the result of `UseHint`. If the result of this operation is equal to c the process succeeded and the signature is verified, if they don't equal the verification failed. See Figure 7.4 for the summary of the process.

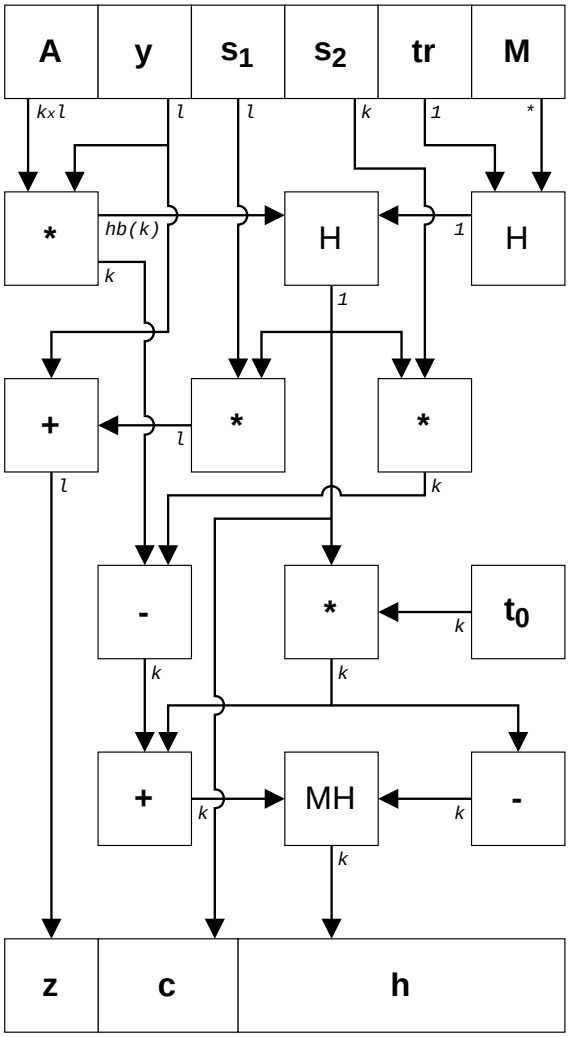


Fig. 7.3: Dilithium signature creation

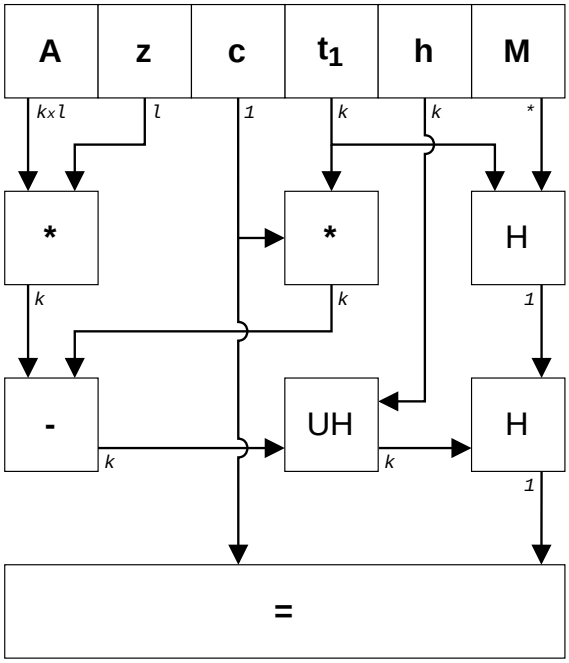


Fig. 7.4: Dilithium signature verification

8 Application capabilities

Many CLI applications programmed in Go use a framework to parse input arguments as options or commands to alter the usage of an application. This framework is called Cobra[1]. Go also has a built-in library for parsing input arguments but it isn't as future rich as Cobra, which contains a lot more useful tools. The following section in this chapter will explain how the Go application in this thesis utilizes it.

The application is mainly split into four groups of capabilities where each group gets a section characterizing it. The parts are

- communication,
- configuration,
- other commands,
- benchmarking.

Command usage examples for some of these capabilities can be found in Appendix B. Additionally, this chapter describes the process adding of new algorithms and extending them. The implementation results of Kyber and Dilithium are also presented in a separate section of this chapter.

8.1 Commands and flags

The Cobra library is able to split an application into logical parts and execution paths, also called commands. A group of commands form a hierarchical structure, which implies that a command can only be used if its parent commands were used beforehand. An example of a group of commands is shown in Figure 8.1.

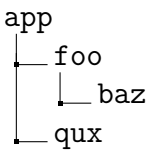


Fig. 8.1: Example command tree

In this case the command `baz` can only be used if `foo` has been used before

`app foo baz`.

On the other hand `baz` can't be used if it doesn't have the required parent commands present, so the Cobra parser would throw an error if given this set of commands

`app qux baz`.

[1] `https://cobra.dev`

Additionally, Cobra allows the programmer to add a flag to a command that alters the command in some way. Flags can be inherited by other commands so that they don't have to be defined in every command separately. A good example of an inherited flag is the `--log` flag which enables a level of logging for the application. As this flag is created in the root of the command hierarchy of this application, all of the subcommands will share it. An example of using a flag other with commands using the previous command structure can look like this

`app foo baz --log info`.

The full list of commands for this application is presented in Figure 8.2. Each command has more flags that alter its execution path. Individual commands and flags can be explored in more detail by using the application. The commands that are greyed out don't alter the execution path by themselves but are needed for creating a parent for its subcommands so that flags can be shared amongst the command children. Another use for them is just having a parent command to logically group the command children. That means if they are run by themselves, the application just prints the output of the `help` command.

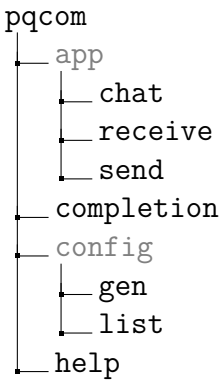


Fig. 8.2: Command tree

8.2 Communication

The main purpose of this application is to create a secure post-quantum communication channel, through which users can send data. The communication mode can be used by invoking the command `app` together with the 3 sub-commands which are described in subsections. Additionally, every subcommand contains 4 shared flags where 3 of the 4 flags are used for altering the addressing, so ports and addresses.

One of the flags is used to alter the configuration of the application (see Section 8.3 for configuration options). A configuration file path can be specified in 3 ways

- environmental variable `PQCOM_CONFIG` – this variable can be set to point to the configuration file,
- the `--config` flag – a relative or absolute path to the configuration file,
- default path – if the above two are not specified a default configuration path will be used, it is listed as the config directory in Appendix E.

8.2.1 Chat command

An interactive mode where users can send text messages asynchronously. By running the application using this command, the TUI (Terminal User interface) is utilized. A TUI is very similar to a GUI (Graphical User interface) where the only difference is that it doesn't require any desktop environment and only requires a terminal interface to work. The created TUI is very easy to control, the user types in a message into the text field and can send that message to the other peer by pressing enter. The sent message will then appear in the window above together with any messages that were sent to him. To quit the application the user can press either escape or ctrl+c. The TUI is also responsive to any window size changes since it is implemented via the bubbletea[2] TUI framework. Examples of light and dark terminal themes can be found in Appendix G.

The underlying application protocol works differently depending on whether the user is a client or a server. The role of the user can be chosen by using either the `-c` flag to act as a user or the `-l` flag to act as the server.

8.2.2 Receive command

In this mode, the application is run in read-only mode and can only display or save sent data. As the data is read it can either be

- redirected through the stdout (standard output) of the terminal to a different command via the pipe (|) operator or redirected to a file via the redirect operator (>),
- saved directly to a new file, by supplying the destination directory where the new files will be created.

To choose one of these options command flags have to be used, if no flag is supplied the data is sent to stdout. By supplying the flag `--dir [directory]` the second option can be used. The supplied directory can be either an absolute or relative path.

[2] `https://github.com/charmbracelet/bubbletea`

8.2.3 Send command

This mode is the opposite of the `receive` command since it is a write-only mode, where the user can send data to another user. Similarly, the user can choose to send data in two ways

- using the output of another command as the input via the pipe (|) operator,
- reading the contents of a file by supping the path to it.

Again the default approach when flags are not supplied is utilizing the first option. The flag `--file [path]` is used for supplying the file path to the input file.

8.3 Configuration

To choose what post-quantum algorithms will the application use, a configuration file is used. It is in a JSON format and 4 keys can be configured

- `kem_alg` – KEM used for key exchange between clients,
- `sign_alg` – digital signature algorithm for creating/verifying signatures during the initial communication,
- `public_key` – base64 encoded string of the public key,
- `private_key` – also a base64 encoded string of the private key.

Similarly, as with the `app` command the `config` command servers to logically separate subcommands. In this case, it's the `list` and `gen` commands. The latter is used for generating a new configuration file with all of the keys filled out and the former is for listing available algorithm names. These algorithms can be seen in Appendix C.

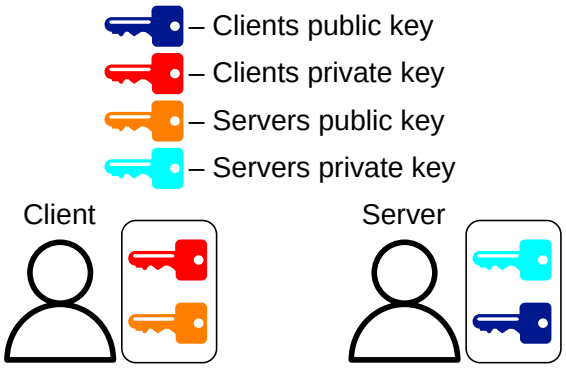


Fig. 8.3: Configuartion file keys

While generating a configuration file the first two keys for the choice of algorithms can be configured using flags `--kem` and `--sign`. By supplying a string parameter to these flags the algorithm name is chosen. The selection of algorithms is generated from the source code which defines the algorithms and their functions. To get a

better understanding of how these algorithms are defined in code, refer to Section 8.5. The other two keys for the public/private key are generated depending on the choice of algorithms. If no algorithms are selected, default algorithms are used. Finally, two configuration files are generated one for the client and one for the server. The client's configuration file contains the public key of the server and his private key. The same can be said for the server where he has the client's public key and his private key (illustrated by Figure 8.3). This is necessary for the initial communication establishment phase of the underlying protocol. See Chapter 9 for more information.

8.4 Completion and help

These two commands are present in any Cobra application by default. The command `help` is self-explanatory and provides information about a given command, like its description and flags. Another default command is the `completion` command which provides scripts that add the autocomplete feature to the application's commands. Autocomplete provides the user with the completion of available commands when pressing the tab key. The installation of the autocomplete functionality depends on the environment. For example if on Linux the running shell is bash, the output of the completion command needs to be copied to the `.bashrc` file. After reloading the terminal autocomplete should now work on the compiled binary.

8.5 Algorithm modularity

Any key encapsulation method or digital signature can be added to this application. Modularity in this application works by implementing methods of a Go interface. An interface serves as a definition of methods, their parameters and return types without actually giving them an implementation. All of the interface methods need to be implemented for an algorithm to be a valid choice. Defined methods for the KEM interface are listed in 8.1. The first three methods are self-explanatory. Method `EkLen` needs to return the size of the public key, `CLen` returns the size of the ciphertext. `Id` needs to return a random number, which is not already returned by any other algorithms ranging from 0 to 255. If an ID is already taken by another KEM, the application will throw an error asking the user to change the ID.

The interface for signatures is shown in Listing 8.2. As with the KEM interface, the signature interface apart from the first three methods also needs a method that returns an ID, private/public key length and the signature length. After implementing these interfaces they need to be added to a shared map contained in

either `crypto/kem.go` or `crypto/sign.go` files. The key for the map entry is the algorithm name that will be used in the configuration file and the value is a pointer to the implemented interface (a Go `struct`). See the listing below for an example.

Listing 8.1: KEM interface

```
type KemAlgorithm interface {
  KeyGen() (puK, prK []byte)
  Dec(c, prK []byte) (key []byte)
  Enc(puK []byte) (c, key []byte)
  EkLen() (ekLen int)
  CLen() (cLen int)
  Id() (id uint8)
}
```

Listing 8.2: Signature interface

```
type SignAlgorithm interface {
  KeyGen() (puK, prK []byte)
  Verify(puK, msg, signature []byte) bool
  Sign(prK, msg []byte) (signature []byte)
  SignLen() (signLen int)
  PuKLen() (pkLen int)
  PrKLen() (skLen int)
  Id() (id uint8)
}
```

Listing 8.3: KEM algorithms map

```
var kems = map[string]KemAlgorithm{
  "PqComKyber512": &kem.PqComKyber512{},
  "CirclKyber512": &kem.CirclKyber512{},
}
```

8.6 Benchmarking

A speed performance benchmark can be run on all of the post-quantum algorithms – digital signatures and key encapsulation methods – that are added to the application using the modularity system. Benchmarking is done using the standard Go library. This approach has many advantages over building a custom benchmarking tool. It is very precise meaning it only measures the actual time it took to run a function. It also integrates very well with another tool called `benchstat`[3]. It is used for

[3] `https://pkg.go.dev/golang.org/x/perf/cmd/benchstat`

summarizing the resulting benchmark and providing a variance to the resulting measurements. The list of available benchmarks can be found in C. A guide on how to benchmark can be found in Appendix B.5.

By default, the go benchmark command runs a function or a piece of code as long as the complete execution time is 1 second. Then it divides the total amount of time it took to execute by the number of iterations ran. The result is the time it took to run one iteration. However, this setting can be changed by appending a parameter. For example in order to run a function for 2 seconds, the parameter

- `-benchtime=2s`

can be provided. One benchmark of a function can also be run multiple times by using the

- `-count=x`

parameter and supplying `x` repetitions. If the number of repetitions is six or more `benchstat` can provide a variance to the resulting time of one iteration. To choose what benchmarks should be run, the parameter

- `-bench=regex`

can be used provided with a regex expression. There is a possibility that the benchmarking/testing processes exists before finishing. This is because the default timeout is 10 minutes. If a benchmark will take longer than 10 seconds the timeout needs to be overwritten by supplying the

- `-timeout=24h`

command. In this example, the timeout is set to 24 hours.

8.7 Optimization process

In order to maximize the performance of Kyber and Dilithium the implementation Go code needs to be optimized. However, before the optimization process can begin a benchmarking process has to be established in order to create a baseline to compare to. The benchmarking process is described in Section 8.6 and was used during the optimization process. The measured speed of algorithms is rarely going to be the same ten times in a row or even two times in a row, especially Kyber and Dilithium, where the speed of calculations depends on the generated keys. Even more when the generation of the keys is dependent on pseudo-randomly generated values which in this case it is. This is why the result is going to be different every time a benchmark is run. So even if a code change that speeds up the algorithm is implemented the result might be worse. To combat this a benchmark is run multiple times in a row to generate more data which then can be used to calculate the mean of those runs. The tool `benchstat` is able to calculate the mean and even the statistical deviance of the mean by using data from a benchmark run. This way it is possible to tell

whether a change in the code was beneficial to the performance of the algorithm or if it was just a statistical fluke.

Now that a proper baseline and a benchmarking process has been established the optimization can begin. Preallocation instead of appending is one of the best ways to optimize a Go program especially Kyber and Dilithium where the program has to iterate over large arrays. When appending to an array in Go, the array size is increased dynamically depending on the number of elements in it. This means every time the size of the array is exceeded a new array has to be allocated and the contents of the old array has to be moved. This takes up a lot of instructions. A better way of appending in this situation is to preallocate the required amount of memory for the array and just insert elements into the array.

Another important thing to keep in mind while optimizing is that division is a much more time consuming operation than others. It is always preferable to use multiplication or bit shifting together with basic binary operations like AND and OR when possible over division. A good example is byte decomposition since this operation is used a lot in Kyber and Dilithium. One of the straightforward solutions might be to divide a number by the powers of 2 up to 8 and then reduce the result modulo two to get the bit in that position (see Listing 8.4).

Listing 8.4: Byte decomposition using division and modulo

```
5 for i = 0; i < 8; i++ {
6   bits[i] = int(num/1<<i) % 2
7 }
```

This approach is slow compared to some other possible approaches because it uses division which is a slow operation. Additionally, the result is converted to an integer which wastes all the instructions that were used to calculate the decimal points of the division result. A better approach would be to first shift the number right for each bit of the number. So for an 8 bit number (a byte) shift the number by the bit position. Then AND the shifted number with a mask of value 0x1. The result will be the bit in the position of the shift count. So for example by shifting the number 5 times and ANDing it with 0x1 the result will be the fifth least significant bit of the number. This implementation can be seen in Listing 8.5. Many more optimizations are used in the implementation of Kyber and Dilithium in this thesis and can be seen by viewing the implementation source code.

Listing 8.5: Byte decomposition using AND and bit shifting

```
1 for i = 0; i < 8; i++ {
2   bits[i] = (num >> i) & 0x1
3 }
```

8.8 Measuring results

The results of the optimization process described in Section 8.7 for booth Kyber and Dilithium can be seen in Tables 8.1 and 8.2. The tables compare the average speed of each algorithm measured using benchmarks for two implementations. The Circl[4] implementation is a cryptography library created by Cloudflare. The implementation of Kyber and Dilithium in this thesis is called PqCom. The process of benchmarking included key generation, encapsulation, decapsulation for Kyber and key generation, signature creation, and verification for Dilithium. The benchmarks were run on an AMD Ryzen 5 3600 processor.

Tab. 8.1: KEMs preformance summary

	PqCom Average [µs]	Circl Average [µs]
Kyber512	$459.9 \pm 1\%$	$109.7 \pm 2\%$
Kyber768	$707.9 \pm 1\%$	$171.7 \pm 2\%$
Kyber1024	$1019 \pm 1\%$	$267.4 \pm 2\%$

Tab. 8.2: Digital signatures preformance summary

	PqCom Average [µs]	Circl Average [µs]
Dilithium2	$1986 \pm 1\%$	$472.9 \pm 1\%$
Dilithium3	$3206 \pm 2\%$	$788.1 \pm 0\%$
Dilithium5	$4130 \pm 1\%$	$1054 \pm 1\%$

More detailed benchmarks can also be found in Appendix D where each table contains the performance information about one implementation of an algorithm, all of its modes and also measurements of its sub algorithms. Each table also contains comparisons to other models of processors. The exact command that was used to benchmark is

- `go test -bench="Kem|Signature" -run=^# -count=20 -timeout=24h \ ./... | tee out.txt`

This command runs benchmarks defined in Appendix C. Each benchmark is run 20 times to reduce the amount of statistical noise. The output is written to the file `out.txt`. This file is then parsed with `benchstat` application mentioned in Section 8.6.

[4] `https://github.com/cloudflare/circl`

9 Network communication and security

The underlying communication protocol that was created for this application is described in this chapter. More specifically the header structure, types of messages and the connection initialization. The second part of this chapter will describe possible approaches to attacking this protocol and techniques that prevent such attacks.

A Wireshark dissector script has been created for the protocol that will be described in this chapter. This is a Lua script that can parse raw data into a defined protocol structure in Wireshark. Examples of captured messages and a guide on using Wireshark together with this Lua script can be found in Appendix F.

9.1 Protocol definition

As with any modern L7 protocol, the messages in this protocol consist of a header the actual data that is being sent also referred to as a payload. The data can either be used for initializing the connection or just for sending arbitrary data. The header is very simple and consists of only two fields the length of the payload and the type of the payload. As can be seen in Figure 9.1, the type is an 8 bit integer and the length is a 16 bit integer.

Fig. 9.1: Protocol header

From the possible 255 types only 4 types of payloads are implemented:

- `ClientInitT` – initialization message for the client side,
- `ServerInitT` – initialization message for the server side,
- `ContentT` – generic data payload type,
- `ErrorT` – error messages.

The initialization together with the client and server init types is detailed in the following Subsection 9.1.1. The rest of the types are mentioned in Subsection 9.1.2. Apart from the headers of the message types all of the data is encrypted using a quantum-resistant symmetric cipher, more specifically AES-GCM with a keys size of 256 b. The GCM block mode also computes a MAC which is used for validating the authentication and integrity of the encrypted message [4].

9.1.1 Initialization

Before the initialization process can start client and server need to share each other's public keys, this can easily be done by first generating a pair of config files with the command `pqcom config gen`. Then one of the config files is moved to a peer that wants to connect to the other peer. Another approach would be to share each other's public key. However, it is very important to share either the public key or the config file out of the band or by using another authenticated and secure communication channel.

After public keys have been exchanged, the process starts with the client sending the `ClientInitT` message which is illustrated by Figure 9.2. This is the biggest message out of the four defined messages. It contains six fields in total.

- **Header** – It serves the same purpose as in any other message, to provide a message type and to dictate the payload length.
- **KEM and Sign Type** – These two fields exist for checking whether the two peers have the same algorithm IDs configured in their configuration files. KEM Type stores the id for the key encapsulation method and the Sign Type stores the digital signature algorithm. The ids that are used in these fields are defined while implementing algorithms via the modularity system (see Section 8.5). Booth of the fields are 1 byte long so for each algorithm type there are 255 possible algorithms.
- **Timestamp** – To prevent repeat attacks, a timestamp is always sent by the client. It contains the Epoch time in microseconds and is defined with a size of 8 bytes. How exactly this timestamp is used to prevent repeat attacks is described in Subsection 9.2.1.
- **Public Encryption Key** – This field enables the server to encrypt a randomly generated symmetric key when he receives the client init message. The client can then decrypt the randomly generated encrypted symmetric key in order to establish a shared key for the symmetric cipher. The length of the public key is defined by the KEM Type field.
- **Signature** – To secure the above-mentioned fields a digital signature is created to protect the authenticity of the whole message. Only the client can create a signature since he holds the private key in his configuration file. The public key was shared with the server beforehand so he can easily check whether a client init message was created by the client. The size of the signature is defined by the Sign Type field.

Once the server receives the client init message it first verifies its signature. Then it checks the timestamp and saves it to a predefined location if nessescery, again to see how exactly this helps prevent the replay attack see Subsection 9.2.1. Next, it

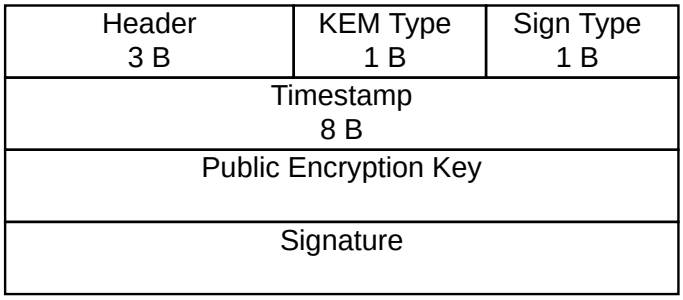


Fig. 9.2: Client inicialization message

checks if the algorithm types are the same as the ones in its configuration file. If all these checks are positive a random symmetric key is generated and encrypted using the public encryption key the client sent. The `ServerInitT` message contains 3 fields as can be seen in Figure 9.3.

- **Header** – Defines message type and length.
- **Key Ciphertext** – Encrypted symmetric key generated by the server. Only the client can decrypt it since the keys were generated by him. There is no fixed length for the ciphertext since its size depends on the KEM type field.
- **Signature** – In order to provide two-way authentication the server has to digitally sign the server init message with the pre-configured private key. The signature can then be verified by the client who has the corresponding public key configured. As with the client init message signature, this signature size also depends on the received and configured Sign Type field.

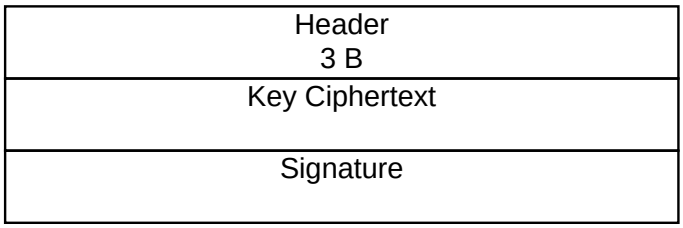


Fig. 9.3: Server inicialization message

Upon receiving the server init message, the client first verifies its signature and decrypts the symmetric key with his private encryption key. Then the encrypted communication can start using a symmetric cipher and the shared symmetric key.

9.1.2 Other communication

The other two message types are relatively simple. The first one is `ErrorT` message which is used for sending error messages. For example in a situation where the client's configured algorithms are not the same as the server's algorithms, the server sends an error message to the client, stating that there has been a misconfiguration. This message type can be seen in Figure 9.4.

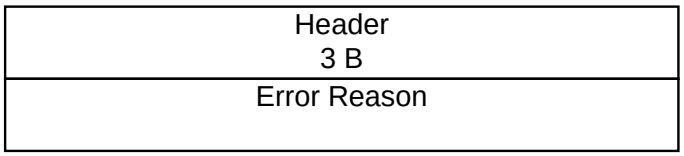


Fig. 9.4: Error message

The last defined message type is the plain data message `ContentT` (see Figure 9.5). This message is used for sending any data that the users want to exchange be it a file or just plain text messages. It can be deduced from the header fields that the maximum payload size is 65 523 or $2^{16} - 1 - 12$ bytes long. The header length is 16 bits long which dictates the maximum payload size to 65535. The nonce used in this message type is 12 bytes long so that has to be deducted from the maximum payload size. This means that in the chat mode, users can exchange a message up to 65 523 bytes long. While in the file sending mode, the file that is being read is read by chunks, these chunks are then sent, so the maximum file size is theoretically infinite. As mentioned beforehand the nonce is also a part of this type of message and is randomly generated for every new message. It is used as the initialization vector for the GCM operation mode while using AES.

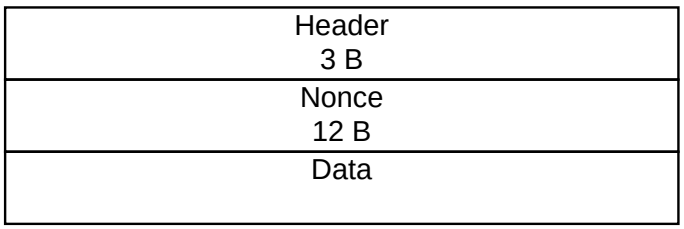


Fig. 9.5: Content message

9.2 Protection against attacks

The most obvious approach to attacking this application would be to eavesdrop on the communication channel and just capture and read the traffic. This is not possible since as mentioned beforehand all traffic apart from the header and initialization messages is encrypted using symmetric cipher AES-GCM-256.

However, more complex approaches could be used for breaking the security of this application. One of those approaches is the impersonation of a communicating entity, in this case, one of the peers. The use of public key cryptography, more specifically digital signatures prevents the use of this approach. An attacker impersonating the client can't create a valid client init message since he can't create a digital signature that would be verifiable by the server. Only the real client can create that signature since he has the private key. Similarly, if the attacker would impersonate the server, he can't create a valid server init message since only the client holds the valid private key to the shared public key.

Another type of attack is the MitM (Man in the Middle) attack. It happens when an attacker manages to create two simultaneous connections between the user and the server. The peers think they are communicating with each other, in truth the attacker is just forwarding their messages back and forth while being able to read them. To prevent this type of attack, the digital signature is again used together with the preconfigured keys. If the attacker wants to create a separate connection with the server after receiving the client init message from the client, the only thing he can do is forward it through to the server, which is harmless. He can't edit it or change the value for example of the public encryption key since he can't create a new signature. He is also unable to impersonate the server since he doesn't know his private key.

9.2.1 Repeat attack

If the attacker would be able to eavesdrop on the communication between two peers let's say, Alice and Bob, he could save the client init message and resend it again at a later time to initialize the connection again. Now Bob would think that he is accepting a connection from Alice, while in truth he's making a connection with the attacker.

To neutralize this type of attack a prevention mechanism is used in the form of timestamp cookies. As mentioned in 9.1.1 the client init message contains a 64 b timestamp. In order to prevent time zone synchronization errors the Epoch time is used. When a server receives a client init, he first creates a name associated with the client's public key hash. If there is no cookie with this name, the server saves

the timestamp in local storage together with the received timestamp. The cookie is always saved to a cookie directory listed in Appendix E. If a new client init message comes in from the same client containing a timestamp that is newer – meaning a higher number, meaning ahead of the current timestamp – the cookie is no longer created just updated with the new timestamp. On the other hand, if the server receives a client init with the same or older timestamp, he drops the connection since the timestamp was not updated with a newer one. This prevents the attack from repeating a client init message. Of course, this works only at the assumption that the first-ever connection from that client is a legitimate one.

Conclusion

Post-quantum algorithms Kyber and Dilithium are successfully implemented in this thesis using the Go programming language. They are also described in this thesis along with other families of post-quantum cryptography. The measured performance of the implementations was compared to a cryptographic library created by Cloudflare. The conclusion is that Cloudflare's implementation of the algorithms is on average about 4.2 times faster than the implementation done in this thesis.

Along with the implementations of Kyber and Dilithium, this thesis also introduces a quantum-safe modular communication application. The modularity is achieved by allowing any post-quantum algorithms implemented in Go to be added with very minimal code changes and good integration into the application. For example, if an algorithm is added it is automatically included in the benchmark suite of tests, where its performance can be compared to other implementations. The final state of the applications uses Kyber which is used to create a session key and Dilithium used for digital signatures. The user of this application can easily change which algorithm he desires simply by editing the configuration file or by generating a new one. Configuration files can be generated by the application and similarly, as with the benchmarking test suites, any added algorithm implementation is seamlessly included as a configuration option.

The underlying protocol which the application uses is very well defined in this thesis and also supports a custom dissector script. This script can be used together with Wireshark to observe the protocol in real traffic scenarios. Methods for preventing common network attacks like Man in the Middle or repeat attacks are also included in the protocol. The application that uses this protocol was designed to be used only in a terminal/console environment. It contains two basic functionalities first it allows the application to send or receive arbitrary data. This data can take a form of a file which can be transmitted between two users. All of this data transfer is of course done over a secured channel. This channel is established using post-quantum cryptography, implied by the aforementioned algorithms. The encryption used during the channel is also quantum-safe since it uses AES-256. The application can also be used as a chatting application. To create a compromise between maintaining a universal text interface and ease of use, the application in this mode runs in a terminal user interface instead of a graphical user interface.

One of the main goals moving forward would be to improve the performance of the affirmation algorithms. To improve the security of the underlying protocol, a formal verification would be required to theoretically prove its security. Additionally implementing Go unit and integration tests would also help strengthen the protocol security.

Bibliography

[1] BERNSTEIN, Daniel J. and Tanja LANGE. Post-quantum cryptography. *Nature* [online]. 2017, 14.9, **2017**(549), 188-194 [cit. 2022-10-09]. Available from: doi:https://doi.org/10.1038/nature23461

[2] SMART, Nigel. *Cryptography: An Introduction* [online]. 3rd. ed. McGraw-Hill College, 2004 [cit. 2020-10-18]. ISBN 978-0077099879. Available from: https://www.cs.umd.edu/~waa/414-F11/IntroToCrypto.pdf

[3] RISTIĆ, Ivan. *Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications Ivan Ristic*. 6 Acantha Court, Montpelier Road, London W5 2QP, United Kingdom: Feisty Duck, 2014. ISBN 978-1-907117-04-6.

[4] PAAR, Christof and Jan PELZL. *Understanding Cryptography: A Textbook for Students and Practitioners*. 2nd edition. London New York: Springer Heidelberg Dordrecht, 2010, 382 s. ISBN 978-3-642-44649-8.

[5] SHANNON, Claude E. Communication Theory of Secrecy Systems. *Bell System Technical Journal*. 1949, **4**(28), 656-715.

[6] BARKER, Elaine and Nicky MOUHA. *Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher*. 2nd ed. NIST Pubs, 2017, 32 s. Available from: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-67r2.pdf

[7] CHEN, Lily, Stephen JORDAN, Yi-Kai LIU, Dustin MOODY, Rene PERALTA, Ray PERLNER and Daniel SMITH-TONE. NISTIR 8105. *Report on Post-Quantum Cryptography*. NIST, 2016, 15 s. Available from: http://dx.doi.org/10.6028/NIST.IR.8105

[8] FIPS PUB 202. *SHA-3 standard: permutation-based hash and extendable output functions*. Gaithersburg, USA: NIST, 2015, 37 s. Available from: http://dx.doi.org/10.6028/NIST.FIPS.202

[9] FIPS PUB 180-4. *Secure Hash Standard*. Gaithersburg, USA: NIST, 2015, 36 s. Available from: http://dx.doi.org/10.6028/NIST.FIPS.180-4

[10] BERNSTEIN, Daniel J., Johannes BUCHMANN and Erik DAHMEN. *Post-Quantum Cryptography*. Berlin: Springer-Verlag, 2009, 248 s. ISBN 978-3-540-88701-0.

[11] YANOFSKY, Noson S. and Mirco A. MANNUCCI. *Qunatum computing for cumputer scientists*. New York: Cambridge university press, 2008, 402 s. ISBN 978-0-521-87996-5.

[12] PITTENGER, Arthur O. *An Introduction to Quantum Computing Algorithms*. Boston: Birkhäuser, 2000, 150 s. ISBN ISBN 0-8176-4127-0.

[13] MCMAHON, David. *Quantum computing explained*. New Jersey: John Wiley & Sons, 2008, 351 s. ISBN 978-0-470-09699-4.

[14] PRETSON, Richard. *Applying Grover-s Algorithm to Hash Functions: A Software Perspective*. Bedford: The MITRE Corporation, 2022. Available from: `https://arxiv.org/pdf/2202.10982.pdf`

[15] MOSCA, Michele. *Cybersecurity in an era with quantum computers: will we be ready?*. Ontario: Cryptology ePrint Archive, 2015, 4 s. Available from: `https://eprint.iacr.org/2015/1075`

[16] IBM Unveils 400 Qubit-Plus Quantum Processor and Next-Generation IBM Quantum System Two. IBM. *IBM Newsroom* [online]. 2022 [cit. 2023-03-20]. Available from: `https://newsroom.ibm.com/2022-11-09-IBM-Unveils-400-Qubit-Plus-Quantum-Processor-and-Next-Generation-IBM-Quantum-System-Two`

[17] GAMBETTA, Jay. Expanding the IBM Quantum roadmap to anticipate the future of quantum-centric supercomputing. IBM. *IBM research* [online]. 2021 [cit. 2022-10-26]. Available from: `https://research.ibm.com/blog/ibm-quantum-roadmap-2025`

[18] ALAGIC, Gorjan, Daniel APON, David COOPER, et al. NIST IR 8413-UPD1. *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*. NIST, 2022, 102 s. Available from: `https://doi.org/10.6028/NIST.IR.8413-upd1`

[19] AJATI, Miklós. Generating hard instances of lattice problems. *Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing* [online]. 1996, 99-108 [cit. 2022-11-01]. Available from: doi:`https://doi.org/10.1145/237814.237838`

[20] GOLDREICH, Obed, Shafi GOLDWASSER and Shai HALEVI. Public-key cryptosystems from lattice reduction problems. *Advances in Cryptology — CRYPTO '97* [online]. Heidelberg: Springer Berlin Heidelberg, 1997, 112–131 [cit. 2022-11-02]. Available from: doi:10.1007/BFb0052231

[21] REGEV, Oded. On lattices, learning with errors, random linear codes, and cryptography. *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing* [online]. 2005, **5**, 84-93 [cit. 2022-11-03]. Available from: doi:10.1145/1060590.1060603

[22] GRIMES, Roger A. *Cryptography Apocalypse: Preparing for the Day When Quantum Computing Breaks Today-s Crypto*. Canada: John Wiley & Sons, 2020, 263 s. ISBN 978-1-119-61819-5.

[23] *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*. 2017, 25 s. Available from: `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf`

[24] BARKER, Elaine. NIST 800-57. *Recommendation for Key Management: Part 1 — General*. 5th edition. National Institute of Standards and Technology, 2020, 171 s. Available from: `https://doi.org/10.6028/NIST.SP.800-57pt1r5`

[25] AVANZI, Roberto, Joppe BOS, Léo DUCAS, et al. *CRYSTALS-Kyber: Algorithm Specifications And Supporting Documentation*. 3rd ed. 43 s. Available from: `https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf`

[26] BAI, Shi, Léo DUCAS, Eike KILTZ, Tancrede LEPOINT, Vaidm LYUBASHEVKSY, Peter SCHWABE, Gregor SEILER and Damien STEHLÉ. *CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation*. 3rd ed. 38 s. Available from: `https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf`

[27] FOROUZAN, Behrouz. *TCP/IP Protocol Suite*. 4th edition. Raghothaman Srinivasan: McGraw-Hill, 2010, 1029 s. ISBN 978-0-07-337604-2.

[28] MARLINSPIKE, Moxie. *The Double Ratchet Algorithm*. 2016, 35 s. Available from: `https://www.signal.org/docs/specifications/doubleratchet/doubleratchet.pdf`

[29] DONENFELD, Jason A. *WireGuard: Next Generation Kernel Network Tunnel*. 2020, 20 s. Available from: `https://www.wireguard.com/papers/wireguard.pdf`

[30] RAYMOND, Eric. *The Art of UNIX Programming*. Addison-Wesley, 2003. ISBN 978-0131429017.

[31] DONOVAN, Alan A. A. and Brian W. KERNIGHAN. *The Go Programming Language*. 2nd edition. Crawfordsville, Indiana: Addison-Wesley, 2016, 399 s. ISBN 978-0-13-419044-0.

[32] AHO, Alfred V., Monica S. LAM, Ravi SETHI and Jeffrey D. ULLMAN. *Compilers Principles, Techniques, & Tools*. 2nd edition. Addison-Wesley, 2006, 1035 s. ISBN 0-321-48681-1.

[33] LIANG, Zhichuang, Shiyu SHEN, Yuantao SHI, Dongni SUN, Chongxuan ZHANG, Guoyun ZHANG, Yunlei ZHAO and Zhixiang ZHAO. Number Theoretic Transform: Generalization, Optimization, Concrete Analysis and Applications. *Information Security and Cryptology* [online]. Springer, Cham, 2021, **12612**, 415-432 [cit. 2022-11-17]. Available from: doi:10.1007/978-3-030-71852-7_28

Symbols and abbreviations

AES	Advanced Encryption Standard
CBD	Centrail Bionimal Distribution
CFB	Cipher Feedback
CLI	Command Line Interface
CRYSTALS	Cryptographic Suite for Algebraic Lattices
CVP	Closest Vector Problem
DES	Data Encryption Standard
DH	Diffie-Hellman
DLP	Discrete Logarithm Problem
DSA	Digital Signature Algorithm
E2EE	End-to-End Encryption
ECB	Electronic Code Book
ECC	Elitic Cruve Cryptography
ECDH	Elitic Curve Diffie-Hellman
ECDSA	Elitpic curve Digital Signature Algorithm
FFT	Fast Furier Transform
GCD	Greatest Common Divisor
GCM	Galois/Counter Mode
GUI	Graphical User interface
IFP	Integer Factorization Problem
IP	Internet Protocol
KDF	Key Derivation Fucntion
KEM	Key Encapsulating Mechanism
KEP	Key Exchange Protocol

LWE	Learning With Errors
MAC	Message Authentication Code
MITM	Man in the Middle
MLWE	Module Learning with Errors
NAT	Network Address Translation
NIST	National Insititue of Standards and Technology
NTRU	N-th degree Truncated Polynomial Ring
NTT	Number Theoretic Transform
OFB	Output Feedback
PDU	Protocol Data Unit
PRNG	Pseudo Random Number Generator
QFT	Quantum Fourier Transform
RSA	Rivest Shamir Adleman
SVP	Shortest Vector Problem
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TUI	Terminal User interface
UDP	User Datagram Protocol

List of appendices

A Lattice-based algorithms diagrams

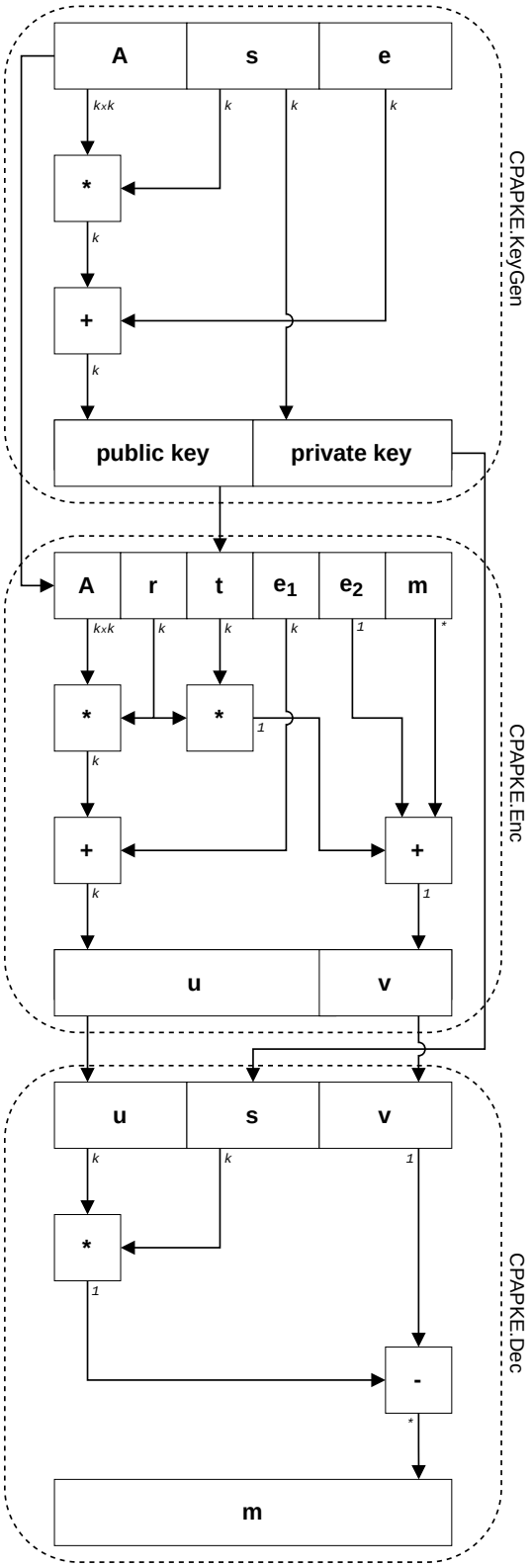


Fig. A.1: Kyber block scheme

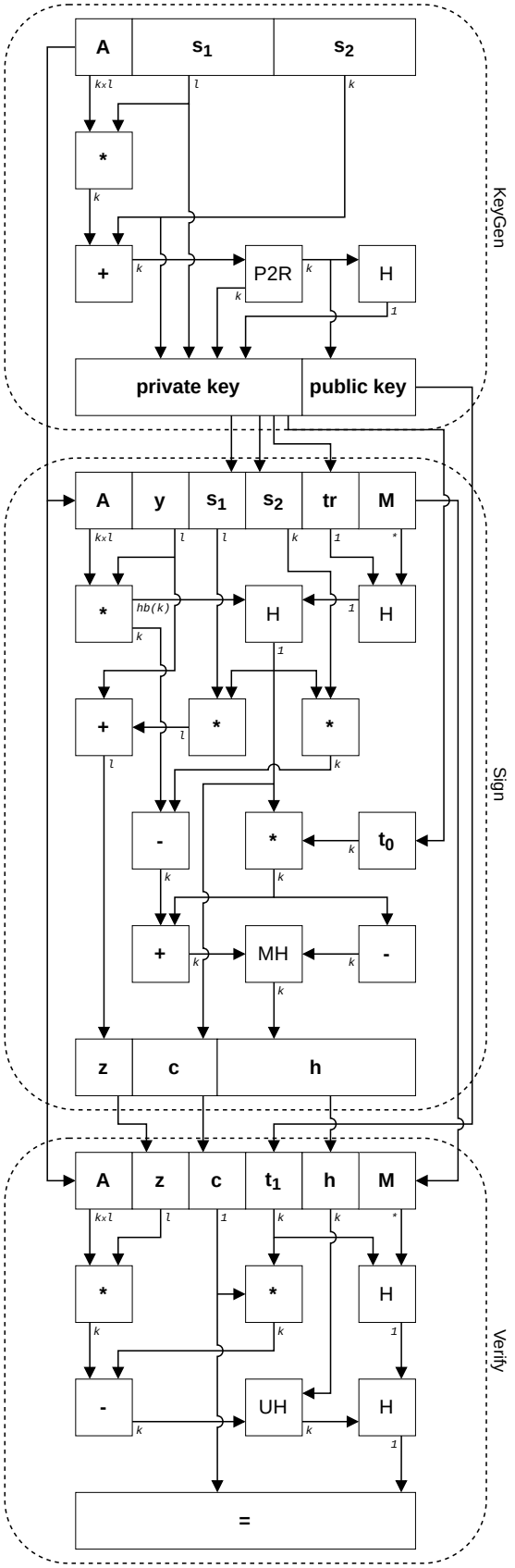


Fig. A.2: Dilithium block scheme

B Go program instructions

This appendix contains the necessary information for building the go program, running it and then instructions on how to run the provided tests.

B.1 How to build

Go supports most of the well-known operating systems such as Linux, Windows and Mac. This application supports the Linux and Windows operating systems. It was not tested on a Mac system but theoretically, it should also work on it. First of all download the latest version of the go binary from this link

- `https://go.dev/doc/install`.

The minimum required Go version is 1.18. Once go is installed the binary executable file for this thesis can be built by running

- `go build -v -o pqcom`

or for Windows

- `go build -v -o pqcom.exe`

inside a command line interface. The command also has to be run inside the root directory of the project. The output of this command should yield a file named `pqcom` or `pqcom.exe` respectively. From now on only the Linux binary will be used for examples but the same examples also apply to the Windows executable.

B.2 How to run

Once the binary is built it can be run just like any other binary. Run with

- `./pqcom`

in the command line interface. To run the Windows executable run

- `pqcom.exe`

Refer to Chapter 8 for application capabilities or run

- `./pqcom --help`

to see what commands are available. Additionally, every command has a short description of what it does on top of available commands or flags. For example to see information about the app command run

- `./pqcom app --help`

B.3 Examples

Before commands that create quantum-resistant connections can be used a configuration file is needed. To generate configuration files for both of the peers run

- `./pqcom config gen`

To use different algorithms for the configuration check the Section 8.3. To use a created configuration file use one of the three options defined in Section 8.2.

In order to receive all log messages the option `--log debug` can be appended to the commands to enable logging at the debug level. By default, the logging will be output to the `stderr` channel of the terminal. However, while the application uses the `chat` mode, log messages are instead saved to a file located in the log directory listed in Appendix E.

B.3.1 Chat mode

Now that configuration files are generated, the application can be used to create connections, for example using the `chat` command. Firstly the server has to start listening. By default command

- `./pqcom app chat -l --config pqcom_server.json`

will start listening on port `4040` and on address `127.0.0.1`. Now a client has to connect by running

- `./pqcom app chat -c --config pqcom_client.json`

where the default remote port is again `4040` with the IP address `127.0.0.1`. If everything was done correctly a TUI should open where users can send messages to each other.

B.3.2 File exchange mode

To run the application in send/receive mode for file exchange one of the peers first has to run

- `./pqcom app receive --config pqcom_server.json > output.txt`

and the other peer has to run

- `cat input.txt | ./pqcom app send --config pqcom_client.json`

When the commands are run in the correct order, the client will send the contents of the `input.txt` file to the server who will save the file in a file named `output.txt`. However, these two commands will only work on Unix-like systems. To create a file exchange between two Windows systems run the following commands. The server side has to run

- `./pqcom app receive --config pqcom_server.json --dir .`

and the client needs to run

- `./pqcom app send --config pqcom_server.json --file input.txt`

The received file will be written in the given directory and named `pqcom_temp_xxxxx` where the five `x` characaters are replaced by random characters.

B.4 How to test

Tests that check whether the implementations are working correctly can be run by entering

- `go test -v ./...`

into the command line interface in the root directory of the project. These tests check whether the implementations of the Kyber and Dilithium are functional.

B.5 How to benchmark

In order to launch benchmarks for all available post-quantum algorithms, meaning all KEM and digital signature algorithms provided via the modularity system, run this command

- `go test -bench="Kem$|Signature$" -run=^# ./... | tee out.txt`

In order to use `benchstat` for a better analysis of the results, first it has to be installed inside the go binary folder by running

- `go install golang.org/x/perf/cmd/benchstat@latest`

In order to use the installed binary the Go binary directory has to be added to the system path environment. The location of Go binary installations is located at `$HOME/go/bin` by default. A new benchmark file has to be generated so that `benchstat` can produce valid statistical data. It can be generated by running

- `go test -count=8 -bench="Kem$|Signature$" -run=^# ./... | \`
  `tee out.txt`

After installing `benchstat` it can be used on the generated file by running

- `benchstat out.txt`

In order to only test the algorithms implemented in this thesis run

- `go test -bench="PqCom\w+All" -run=^# ./... | tee out.txt`

Similarly as before the count be increased so that the results can be analyzed using `benchstat`.

C Available algorithms and benchmarks

List of available algorithms that can be configured in this application taken from the `pqcom config list` command. Algorithms prefixed with `PqCom` are implementations described in this thesis. The prefix `Circl` denotes algorithms from the Cloudflare cryptography library.

```
Key encapsulation methods
├─ PqComKyber512
├─ PqComKyber768
├─ PqComKyber1024
├─ CirclKyber512
├─ CirclKyber768
├─ CirclKyber1024
Digital signatures
├─ PqComDilithium2
├─ PqComDilithium3
├─ PqComDilithium5
├─ CirclDilithium2
├─ CirclDilithium3
└─ CirclDilithium5
```

A list of available benchmarks can be found below where every benchmark from a category can be applied to any algorithm or any algorithm security level included in that category.

```
Key encapsulation methods
├─ Key Generation
├─ Encapsulation
├─ Decapsulation
└─ Key Generation+Encapsulation+Decapsulation
Digital signatures
├─ Key Generation
├─ Signature creation
├─ Verification
└─ Key Generation+Signature creation+Verification
```

D Performance

Tab. D.1: Processor details

Name	Full name	Base Clock [GHz]	Max. Boost Clock [GHz]
AMD 3600	AMD Ryzen 5 3600	3.6	4.2
Intel 2300	Intel Core i5-2300	2.8	3.1
Intel 10610u	Intel Core i7-10610u	1.8	4.9

Tab. D.2: PqCom Kyber performance

Processor	KeyGen [µs]	Enc [µs]	Dec [µs]	All [µs]
Kyber512				
AMD 3600	$115.6 \pm 0\%$	$151.0 \pm 0\%$	$205.4 \pm 0\%$	$459.9 \pm 1\%$
Intel 2300	$235.2 \pm 1\%$	$302.9 \pm 1\%$	$421.2 \pm 0\%$	$948.2 \pm 1\%$
Intel 10610u	$96.65 \pm 10\%$	$92.27 \pm 2\%$	$132.5 \pm 1\%$	$277.9 \pm 1\%$
Kyber768				
AMD 3600	$185.0 \pm 1\%$	$237.8 \pm 0\%$	$309.3 \pm 1\%$	$707.9 \pm 1\%$
Intel 2300	$368.8 \pm 0\%$	$455 \pm 1\%$	$618.1 \pm 0\%$	$1442 \pm 1\%$
Intel 10610u	$138.6 \pm 7\%$	$144 \pm 1\%$	$191.1 \pm 0\%$	$420.3 \pm 1\%$
Kyber1024				
AMD 3600	$277.3 \pm 1\%$	$318.5 \pm 1\%$	$433.9 \pm 0\%$	$1019 \pm 1\%$
Intel 2300	$547.9 \pm 0\%$	$659.6 \pm 0\%$	$863 \pm 1\%$	$2074 \pm 0\%$
Intel 10610u	$250.2 \pm 17\%$	$209.8 \pm 1\%$	$268.4 \pm 0\%$	$607.2 \pm 1\%$

Tab. D.3: Circl Kyber performance

Processor	KeyGen [µs]	Enc [µs]	Dec [µs]	All [µs]
Kyber512				
AMD 3600	$33.9 \pm 2\%$	$39.6 \pm 1\%$	$39.3 \pm 1\%$	$109.7 \pm 2\%$
Intel 2300	$88.98 \pm 2\%$	$102.4 \pm 2\%$	$111.8 \pm 4\%$	$297 \pm 2\%$
Intel 10610u	$19.4 \pm 10\%$	$22.4 \pm 0\%$	$21.2 \pm 1\%$	$60.5 \pm 1\%$
Kyber768				
AMD 3600	$56.2 \pm 2\%$	$63.5 \pm 0\%$	$61.9 \pm 0\%$	$171.7 \pm 2\%$
Intel 2300	$154.2 \pm 1\%$	$165 \pm 3\%$	$182.2 \pm 4\%$	$494.9 \pm 3\%$
Intel 10610u	$31.8 \pm 6\%$	$35.1 \pm 1\%$	$33.3 \pm 0\%$	$89.2 \pm 1\%$
Kyber1024				
AMD 3600	$83.7 \pm 1\%$	$92.02 \pm 0\%$	$88.7 \pm 0\%$	$267.4 \pm 2\%$
Intel 2300	$233.2 \pm 2\%$	$245.7 \pm 1\%$	$265.9 \pm 2\%$	$727.8 \pm 3\%$
Intel 10610u	$46.1 \pm 7\%$	$49.3 \pm 0\%$	$48.1 \pm 1\%$	$130.1 \pm 1\%$

Tab. D.4: PqCom Dilithium performance

Processor	KeyGen [µs]	Sign [µs]	Verify [µs]	All [µs]
Dilithium2				
AMD 3600	$335.4 \pm 0\%$	$1299 \pm 2\%$	$337 \pm 0\%$	$1986 \pm 1\%$
Intel 2300	$661.6 \pm 0\%$	$2996 \pm 5\%$	$705.8 \pm 0\%$	$4445 \pm 2\%$
Intel 10610u	$292.6 \pm 22\%$	$1009 \pm 3\%$	$234.9 \pm 7\%$	$1528 \pm 4\%$
Dilithium3				
AMD 3600	$555.7 \pm 0\%$	$2066 \pm 3\%$	$520.1 \pm 0\%$	$3206 \pm 2\%$
Intel 2300	$1073 \pm 0\%$	$4980 \pm 3\%$	$1085 \pm 0\%$	$7072 \pm 1\%$
Intel 10610u	$498.2 \pm 10\%$	$1842 \pm 11\%$	$403 \pm 7\%$	$2518 \pm 4\%$
Dilithium5				
AMD 3600	$809.5 \pm 0\%$	$2461 \pm 2\%$	$803.9 \pm 0\%$	$4130 \pm 1\%$
Intel 2300	$1596 \pm 1\%$	$5860 \pm 6\%$	$1656 \pm 0\%$	$9060 \pm 3\%$
Intel 10610u	$633.2 \pm 26\%$	$2247 \pm 11\%$	$619.4 \pm 5\%$	$3358 \pm 10\%$

Tab. D.5: Circl Dilithium performance

Processor	KeyGen [µs]	Sign [µs]	Verify [µs]	All [µs]
Dilithium2				
AMD 3600	$99.6 \pm 0\%$	$291.9 \pm 1\%$	$84.3 \pm 1\%$	$472.9 \pm 1\%$
Intel 2300	$344.4 \pm 1\%$	$904.0 \pm 3\%$	$230.1 \pm 3\%$	$1455 \pm 4\%$
Intel 10610u	$60.44 \pm 5\%$	$137.8 \pm 2\%$	$43.3 \pm 1\%$	$235.8 \pm 1\%$
Dilithium3				
AMD 3600	$189.5 \pm 0\%$	$467.6 \pm 1\%$	$135.2 \pm 0\%$	$788.1 \pm 0\%$
Intel 2300	$587.5 \pm 2\%$	$1546 \pm 3\%$	$448.9 \pm 1\%$	$2530 \pm 4\%$
Intel 10610u	$139.2 \pm 6\%$	$215.5 \pm 1\%$	$69.1 \pm 1\%$	$384 \pm 1\%$
Dilithium5				
AMD 3600	$256.7 \pm 0\%$	$574.9 \pm 1\%$	$217.7 \pm 0\%$	$1054 \pm 1\%$
Intel 2300	$891.9 \pm 1\%$	$1994 \pm 2\%$	$736.9 \pm 1\%$	$3733 \pm 4\%$
Intel 10610u	$172.2 \pm 9\%$	$269 \pm 1\%$	$110.2 \pm 0\%$	$509.6 \pm 1\%$

E Directories

This application uses some fixed directory paths to store or load files. Their location depends on the operating system the application is running on. The directories that are used on Unix-like systems are:

- config – `$HOME/.config/pqcom`
- cookie – `$HOME/.cache/pqcom`
- log – `$HOME/.local/state/pqcom`

On Windows systems these directories are:

- config – `$HOME\.config\pqcom`
- cookie – `$HOME\.cache\pqcom`
- log – `$HOME\.pqcom\logs`

F Wireshark integration

To run Wireshark together with the custom Lua script for dissecting the protocol defined in this thesis run

- `wireshark -X lua_script:wireshark/dissector.lua`

while in the root directory of the project. However, this dissector only works if the application is using the default port `4040`. Below are some captured examples.

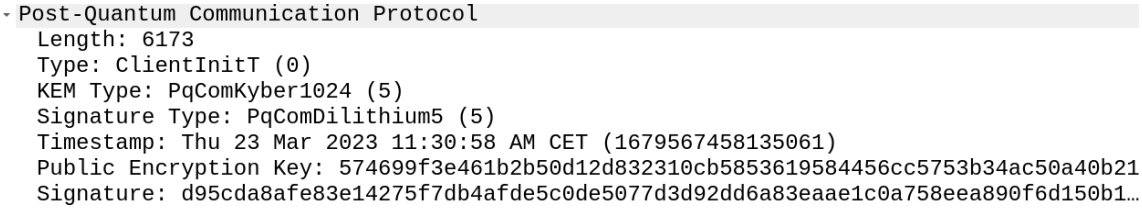


Fig. F.1: Captured client init

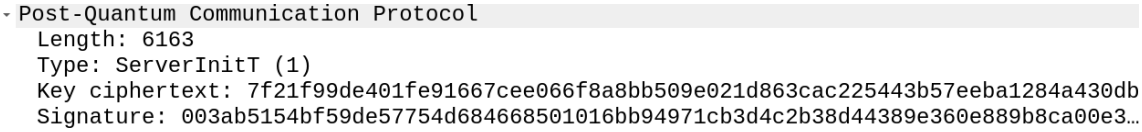


Fig. F.2: Captured server init

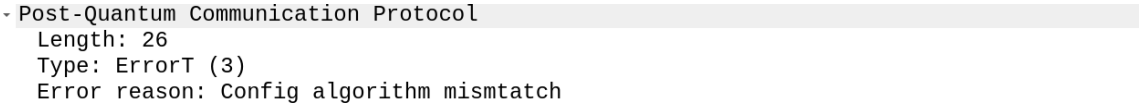


Fig. F.3: Captured error

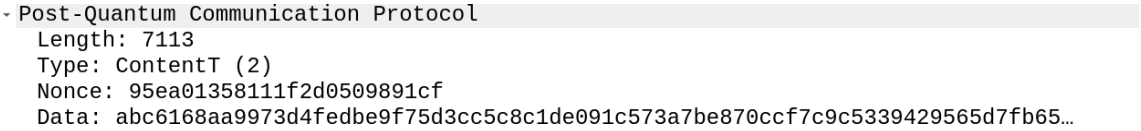


Fig. F.4: Captured data

G Application TUI

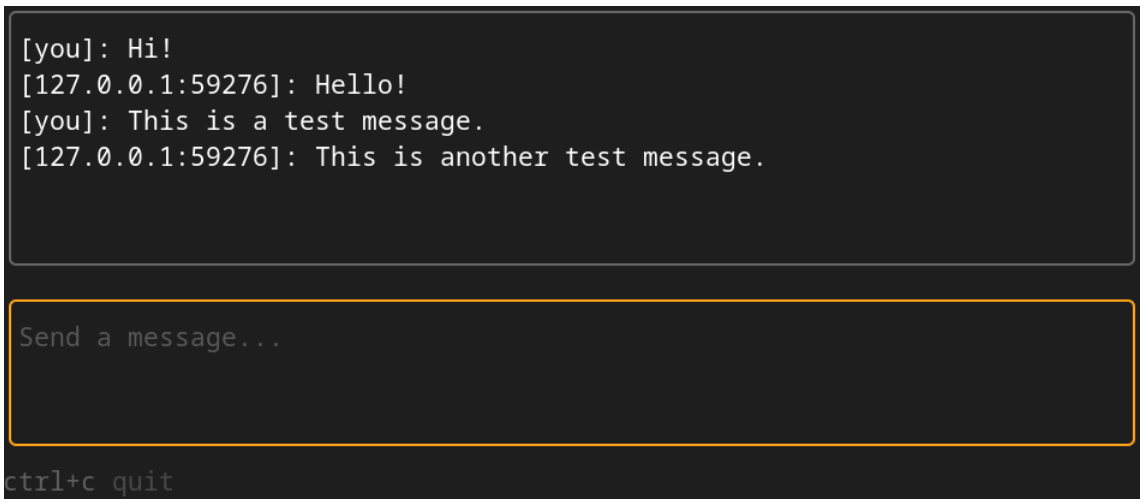


Fig. G.1: Dark theme application TUI

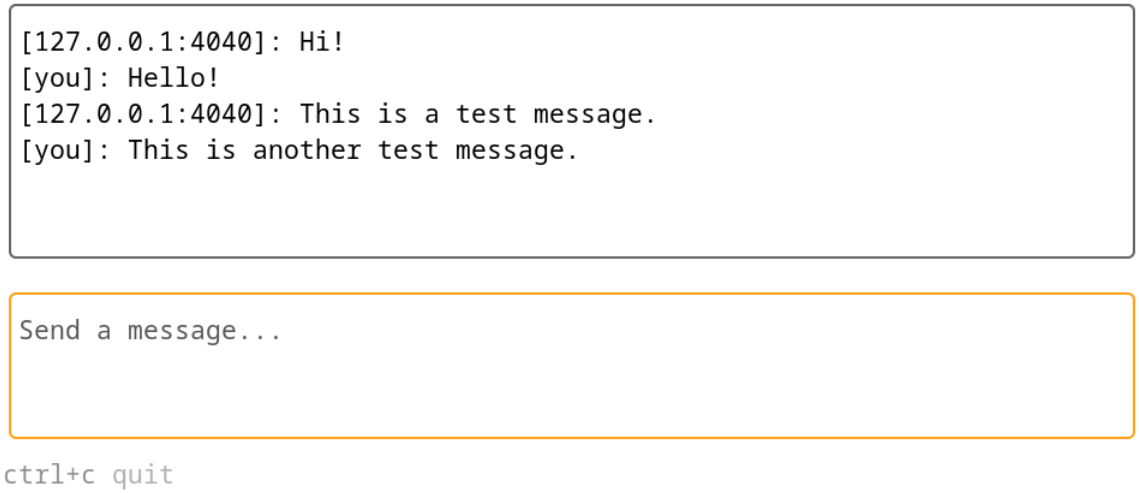


Fig. G.2: Light theme application TUI