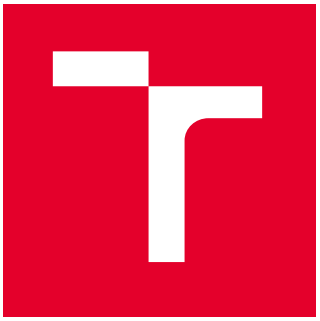
BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

SEMESTRAL THESIS

Brno, 2022

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

MODULÁRNÍ KOMUNIKACE POSTAVENÁ NA POSTKVANTOVÉ KRYPTOGRAFII

SEMESTRAL THESIS
SEMESTRÁLNÍ PRÁCE

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

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

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Semestrální práce

magisterský navazující studijní program **Informační bezpečnost**

Ústav telekomunikací

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

Ročník: 2 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Modulární komunikace postavená na postkvantové kryptografii

POKYNY PRO VYPRACOVÁNÍ:

Téma je zaměřeno na implementaci knihovny obsahující postkvantové algoritmy pro výměnu klíče, šifrovaní pomocí veřejného klíče a digitální podpis. Konkrétní výběr algoritmů pro implementaci a volba implementačního jazyka je na volbě studenta po předchozí konzultaci. Řešitel musí v rámci semestrální práce implementovat aspoň jeden algoritmus z vybrané kategorie. Výstupem SP bude funkční implementace algoritmu obsahující výkonové testování a porovnání s dostupným řešením.

Student v navazující diplomové práci implementuje vždy další jeden algoritmus z každé kategorie a pomocí vytvořené knihovny implementuje komunikaci klient-server, která bude využívat výhradně postkvantovou kryptografii. Klientská část aplikace bude obsahovat konsolové rozhraní, které bude umožňovat stáhnutí a nahrání souborů a výměnu zpráv s ostatními uživateli aplikace.
Dále student navrhne kompatibilní rozhraní, pomocí kterého je možné do aplikace klienta a serveru snadno přidat další postkvantové algoritmy. Práce bude taktéž obsahovat porovnání rychlosti a výkonnosti s ostatními implementacemi postkvantových algoritmů ve vybraném jazyce.

DOPORUČENÁ LITERATURA:

podle pokynů vedoucího práce

Termín zadání: 1.10.2022 ***Termín odevzdání:*** 12.12.2022

Vedoucí práce: Ing. David Smékal

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

Author's Declaration

Author:	Bc. Samuel Kopecký
Author's ID:	211799
Paper type:	Semestral Project
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.

Contents

List of Figures

List of Tables

Listings

Introduction

Development of quantum computers is the next big step in technology evolution and brings many new possibilities of improvement but also many dangers to modern cryptography algorithms. In the first chapter this thesis aims to provide a basic understanding of modern cryptography and an overview of cryptography primitives utilized within it. Next it introduces the reader at a very basic level to quantum mechanics and its basic principles such as quantum superposition. The provided knowledge is then expanded with quantum algorithms such as Shor's algorithm which is capable of braking modern cryptography.

To resolve the problem of a possible quantum supremacy happening this thesis introduces the concept of post quantum cryptography which consists of cryptographic algorithms that are resistant to attacks using quantum algorithms or classical algorithms. One such group of post quantum algorithms are 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 has ended and the two of the winning algorithms Kyber and Dilithium are described.

Part of the practical part of this thesis contains the implementations of these algorithms in the programing language Go. Go was chosen because it creates a good balance between performance and simplicity. The performance is owned due to the fact that is a compiled language, and thats why it can compete with the programing language C. During the description of the implemented lattice-based algorithms some code snippets are also shown. For the sake of readability they don't contain pure Go code but pseudocode styled like Go. To better explain implementations of these algorithms, simplified block diagrams explaining the processes of Kyber and Dilithium are located in appendix A.

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 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 is able to solve the problem without the knowledge the secret in polynomial time, the underling 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 underling 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 confidentiality of the data that is being transferred over a communication medium. The general idea of symmetric ciphers is that they are fast ciphers (compared to asymmetric ones, see section 1.3) 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 with figure 1.1. In a situation where *Alice* wants to send *Bob* an obfuscated document (plaintext), *Alice* firstly 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.

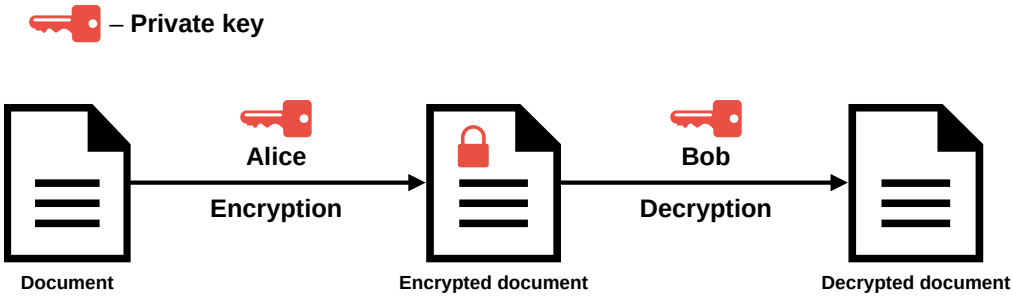


Fig. 1.1: Simplified symmetric cipher

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 receivers knowledge. Some examples of secure mode of operations for block ciphers are [4]

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

These ciphers are based on a substitution-permutation network (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 relation 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 (see figure 1.2). 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 then just a simple substitution-permutation network. Good examples of block ciphers that use this principe 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].

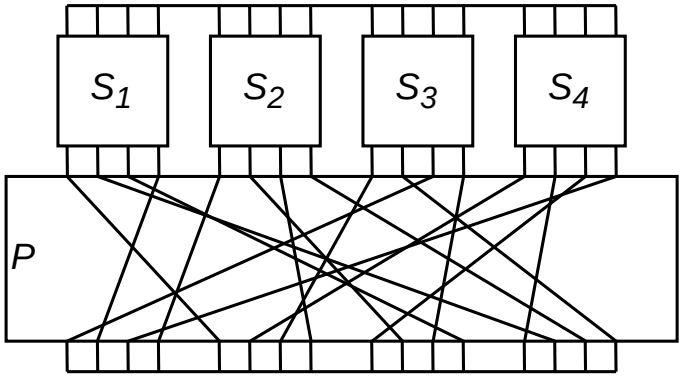


Fig. 1.2: Substitution-permutation network

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 the randomly generated by the PNRG function. Then the message is XORed with generate key. The XOR operation can also be rewritten as mod 2 and thus the encryption process can be described with the the equation

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

and the decryption process

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

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 a set bits into another set of bits:

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

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 posses 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 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 the KECCEK construction also called the sponge construction. The main idea behind the sponge construction is that after each round of compression, a number of bits are firstly absorbed by the compression function and then some bits are taken out of each compression iteration that make up the final hash value. How many bits are absorbed or taken out is dictated by the hash function parameters. [9]

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 ares 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 is able to be shared is called the public key, the second key that has to be kept secret is called the secret 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 principe of an encryption algorithm can be seen in figure 1.3. *Alice* can encrypt a document using *Bobs* 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 secret key since he is the only one that owns it. [2]

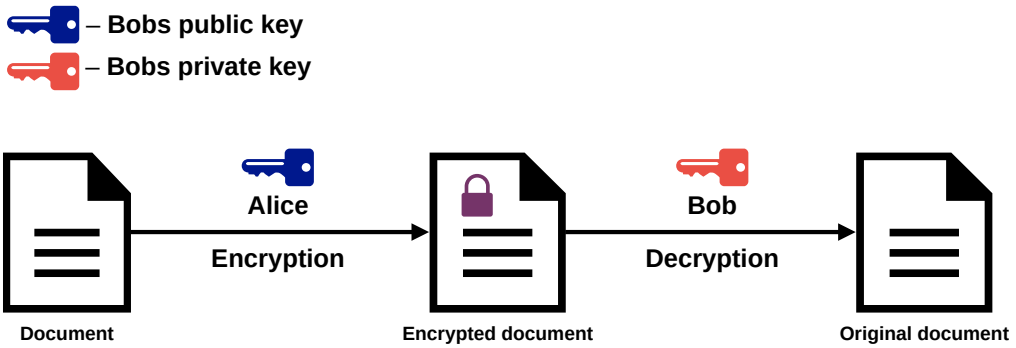


Fig. 1.3: Symplified asymmetric encryption cipher

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 secret key. Everyone else including *Alice* can check whether the document came from *Bob* by verifying the signature with his pubic 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 a the unsigned document. *Alice* would 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 a some data servers the same purpose. Hash functions are described in the previous section of this chapter (section 1.2).

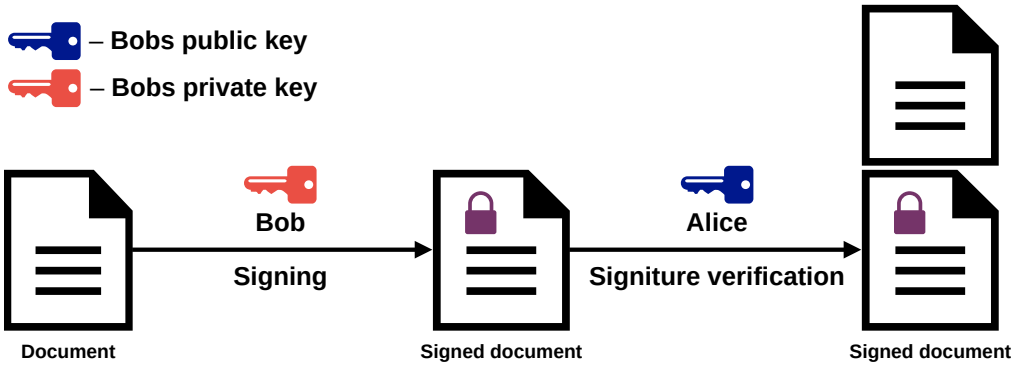


Fig. 1.4: Simplified digital signature scheme

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 (up to 15360 bits) is impossible to compute on todays 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 number. 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 secret and public keys would be derived from the integer n.

The other principle that is used often in todays 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 the modulo operation together with any other operation 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 todays 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 [4]. This problem only works if modulus p is in the bigger then 2048 bits. DSA (Digital Signature Algorithm) utilizes this problem for creating digital signatures. An alternative exists

to this problem 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 secret key [3]. This property is 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 firstly need to be shared between the communicating entities before any encryption can begin. That is where a KEP (Key Exchange Protocol) is utilized. A category 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 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, secret key pair like RSA, but each entity exchanges their public key with the other entity and then they calculate the secret key from the knowledge of their own secret key and of the opposite entities 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 then plain DH or RSA [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 at ordinary sizes, they can exist in more then 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 we measure an object in such a state the position of the object collapses into a single point in space. [11]

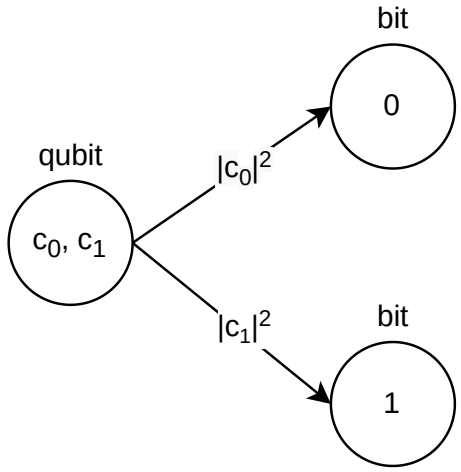


Fig. 2.1: Representation of a qubit

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

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

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.

Using complex numbers qubits can also 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 (0 and 1). A quantum computer can hold more then one quibt 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. A quantum algorithm doesn't work like a classical algorithm. It starts of with a single position for all the qubits in the input. During the algorithm, the quibts 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 thread 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 (subsection 1.3.1) used in RSA, which can now 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 then on a quantum computer.

The first part of the 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

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

fortunately we can use the 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 order of a is computationally infeasible in polynomial time using a classical computer, thats why a quantum computer is needed to find r and will be explained later. If r is odd or $a^r \equiv -1 \bmod N$ it is discarded and and a new r is found. Equation 2.4 can now be altered by subtracting 1 from booth 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 also be used for breaking the DLP. [11][13]

The second part of the algorithm as mention is used to find the order of a. An order of a number can also be represented as a period of 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 mention 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. [12]

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 a binary string 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 output a binary string also 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 the the it finds a match it can retrospectively find the output that generated the hash value [14]. Since the Grover's algorithm is not as efficient as Shor's algorithm in find solutions that break ciphers or algorithms, key/hash value sizes can be increased to prevent theses kind 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 sufficient amount of qubits using the the 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 127-qubit quantum processor in 2021 so humanity is not yet at the point where everyday internet communication using public key cryptography can be broken using quantum computers [16].

Although that doesn't change the fact that future quantum computers might be able to. It fact IBM has projected in their new roadmap to a practical quantum computer, that by 2026 they expect to have working quantum computers that contains between 10 000 and 100 000 qubits [17]. Still a replacement for the current public key algorithms needs to be found. Each of the new candidates will be discuses 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 happened in the future. New public key algorithms need to be standardized so they can be used as replacements for the quantum vulnerable algorithms such as RSA and ECDH. NIST (National Insititue of Standards and Technology) has began the first standardization process for post-quantum algorithms, which are algorithms that are resistant to the future thread of quantum computers [7]. The main candidates that will be described in individual subsections are

- Lattice-based cryptography,
- Code-based Cryptography,
- Multivariate Public Key Cryptography,
- Hash-based digital signature schemes.

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 3.2) [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 bases for the lattice [19]. Linearly independent vectors have the special property of not being a combination of any other vectors from the set of linearly independent vectors. An example of these vectors is also 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

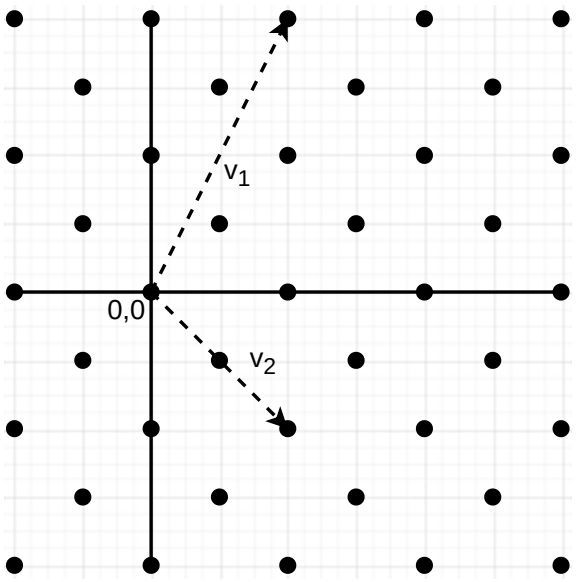


Fig. 3.1: 2-dimensional lattice

and calculate back the input that was given, it other words a one-way function has to exists. 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 1.4 KEM (Key Encapsulating Mechanism) is a 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 secret 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 the called the bad basis which

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 not 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 $\mathcal{L}$. Next a small noise vector e is chosen that is not a lattice point in $\mathcal{L}$. 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]

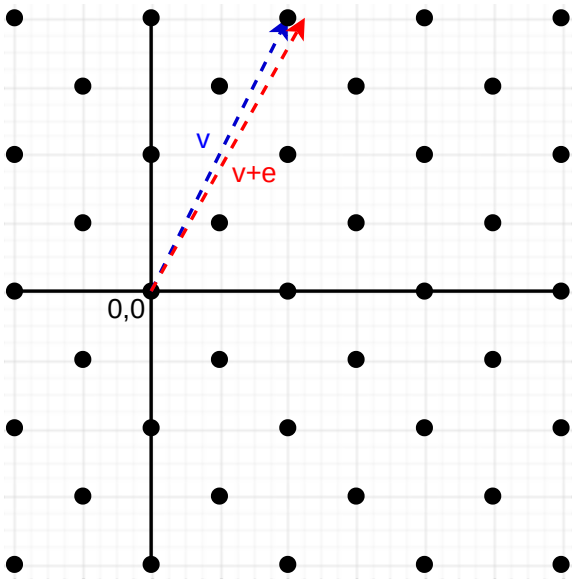


Fig. 3.2: Closest Vector Problem

Finding the original vector from the cipertext 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 fact that the CVP is easily computed while using the good basis B, but hard in the bad basis H. As mention 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 (secret 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 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 p-coefficient polynomial:

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

Like GGH this scheme is based on the CVP. The message consists of two polynomials d, e which are not a lattice points and the ciphertext c which is calculated from d, e where h points to a lattice point. An attacker cannot easily compute back d, e only with the knowledge of h, c. The use of polynomials makes NTRU much faster then the GGH cryptosystem and was considered heavily for post-quantum standardization by NIST [1].

LWE is not a cryptosystem by itself, but many cryptosystem are based on it. The problem is based in 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)

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

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 3.2).

3.1.3 Digital signature schemes

As described in 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 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 3.2.

3.2 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 of 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 4 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

Tab. 3.2: Forth round submissions [18]

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

4 Network basics

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

The TCP/IP protocol suit consists of 5 layers as can be seed 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 [23]. 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 horizontally with other protocol layers using PDUs (Protocol Data Unit), each PDU is either encapsulated or de-encapsulated into another PDU, depending on 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 fot that specific layer. For example the IP address is contained in the header for the network layer. The layers also communicate vertically using either physical channels (physical and data link layers) or virtual channels (all other layers). Physical channels are created between a physical medium by which the bits travels through, virtual channels on the other hand are created between applications on devices. Channels on each layer use a different identifier to differentiate between them. Aforementioned information for each layer can be found in table 4.1.

Tab. 4.1: TCP/IP protocol suite [23]

(#) 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 applications 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 [23]. 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 is able to listen on the same port for booth TCP and UDP protocols at the same time.

The UDP protocol was designed to be fast and unreliable [23]. It posses 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 size. These are then sent one by one to the targeted entity.

The TCP protocol on the other hand is a connection-oriented protocol. Before data can transferred between two entities first a connection has to be established using the three-way handshake (figure 4.1) [23]. It works by setting one 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 is able to 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 [23]. 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 then UDP. An example of good usage for the TCP is the HTTP protocol where a website needs to transferred exactly as intended without any errors.

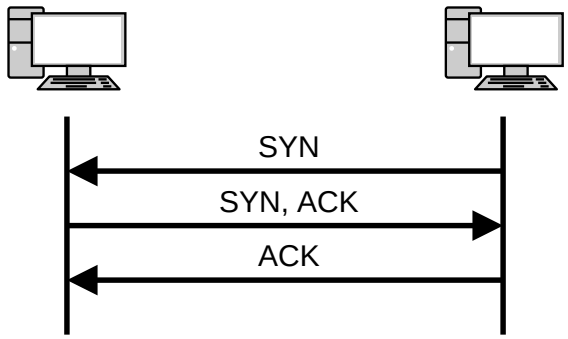


Fig. 4.1: Three-way handshake

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 the former is more commonly used in the internet [23]. As the name implies a service is hosted on a server, 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 were also seen by the server, which implies the users have to trust the server to not store or log their exchanged messages anywhere.

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 also since each entity needs to be able to listen for incoming connections and also accept create connections to other listening entities.

The issue with using peer-to-peer for communicating with users, is that each user needs to have an open port. But since most of 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 mention before this leaves the messages open and readable by the server.

5 Implementing Lattice based cryptography

As mentioned in 3.2 three algorithms have been standardized that are based on lattice cryptography. This chapter will describe two of these algorithms in more detail, namely CRYSTALS-Kyber and CRYSTALS-Dilithium. Their respective sections will also describe some implementation aspects of these algorithms in the programming language Go. Go is a relatively new programming language compared to others (2009) as of writing this [24] and its detailed characterization will follow in the next section of this chapter. This thesis will in the future also contain a client-server application for exchanging messages also programmed in Go and will utilize only quantum resistant algorithms.

5.1 The Go programming language

The first iteration of the Go programming language was created at Google. It is an open-source programing language and has many similarities with C which means it is a compiled language and a statically typed language [24]. 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 be ran multiple times without the need to compile again [25]. This makes Go faster then most of interpreted languages like Python, since an interpreter needs to translate the source code every time it has to run.

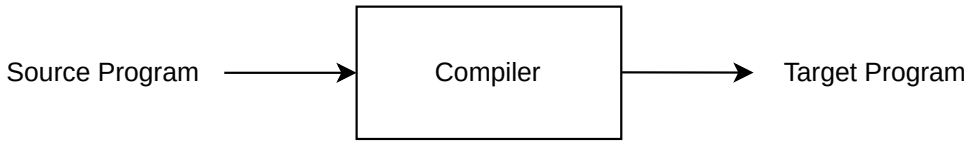


Fig. 5.1: Compiler

However unlike C it has a garbage collector, that means it has an automatic memory management [24]. In C a programmer has to manage memory on his own, allocate and free it by using functions. Go and its garbage collector take 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 thats why it was chosen for this thesis.

5.2 CRYSTALS-Kyber

CRYSTALS (Cryptographic Suite for Algebraic Lattices) Kyber is a quantum resistant KEM (Key Encapsulating Mechanism) standardized in the 3rd round of the NIST standardization process. It is based on modified version 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 in 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]

5.3 Implementing Kyber

The implementation of Kyber in this thesis is done using only the standard Go library with the exception of one external library [26] which is required for the implementations of the hash functions `SHAKE-128/256` and `SHA-3` (refer to section 1.2 for hash functions). Figure A.1 illustrates a very simplified block diagram of how Kyber works. The small letter or a number at the start of an arrow denotes the size of the object in polynomials. So for example the letter k denotes that an object consists of k polynomials. At first the public and secret key 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. The following subsections will explain each sub algorithm of the block diagram in more detail along with code snippets. However the code snippets are only a go styled pseudocode and can't be actually compiled, check the practical part of this this for the compilable go source code.

Kyber uses a set of parameters to define its security level, of which it has three as seen in table 5.1. Kyber in this thesis is only implemented for the parameters Kyber512. What individual parameters mean will be explained in further subchapters.

5.3.1 Theoretical background

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

$$\mathbb{Z}_q[X]/(X^n + 1). \tag{5.1}$$

Tab. 5.1: Kyber security levels [27]

	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	$(11, 5)$

A ring in 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 5.1. An example of a polynomial with elements from the ring R_q

$$t_1 = 1564 + 2189x + 258x^2 + \cdots + 655x^{n-2} + 2587x^{n-1}. \tag{5.2}$$

A vector of size k consists of k polynomials with coefficients from the ring R_q. The parameter k can again be found in table 5.1. The the polynomial t_1 from the previous example together with a new polynomial t_2

$$t_2 = 2408 + 1932x + 420x^2 + \cdots + 3256^{n-2} + 2399^{n-1}, \tag{5.3}$$

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. [27]

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 $n \log(n) = 1387$. 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 $\hat{\mathrm{A}}$. [28]

5.3.2 Encoding, Compression and randomness

In order 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 `poly` into $32 * l$ bytes,
- `decode`(B, l) – convert $32 * l$ bytes into a polynomial.

In the following code listings functions like `encodePolyVec()` exist. It encodes every polynomial in the vector. The same applies for the decode functionality.

Another feature of Kyber is the compression of polynomials that are encoded. Due to the fact that Kyber is based on LWE, the calculations don't need exact numbers to be correct. This is why a compression mechanism that discards some low-order bits from encoded polynomials can be introduced. 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.

Similarly to the encoding/decoding function theses functions can be applied for every coefficient of a polynomial.

Whenever it mentioned that a random polynomial or a polynomial vector has been generated it is implied that a CBD (Centrail Bionimal Distribution) is used with a parameter either η_1 or η_2.

5.3.3 Key generation

The key generation functions starts with a function `cpapkeKeyGen()`. At first parameters need to be generated, this process is described in listing 5.1. At first two 32 B arrays (ρ and σ) are created at random. Matrix $\hat{\mathrm{A}}$ of $k \times k$ polynomials is then generated from ρ with the function `genPolyMat()`. This function is deterministic, so for the same values of ρ the function will always generate the same matrix. The matrix $\hat{\mathrm{A}}$ is publicly know to everyone and needs to shared, but since the function that generates it is deterministic only ρ needs to be shared instead of the while matrix. The generation function also makes sure the matrix is already in NTT form. Next the polynomial vectors $\hat{\mathrm{s}}$ and $\hat{\mathrm{e}}$ are generated at random using σ and transformed into the NTT domain.

Listing 5.1: Key generation variable setup

```
4   rho, sigma := randBytes(64)
5   A_hat := genPolyMat(rho)
6   s_hat := nttPolyVec(randPolyVec(sigma))
7   e_hat := nttPolyVec(randPolyVec(sigma))
```

After variables have been setup the actual key generation can begin (listing 5.2). The vector $\hat{\mathrm{t}}$ is calculated by multiplying $\hat{\mathrm{A}}$ by $\hat{\mathrm{s}}$, and then increased by adding the vector $\hat{\mathrm{e}}$. The public key is then generated from the the encoded vector $\hat{\mathrm{t}}$ and the ρ byte array. The public key is just the vector $\hat{\mathrm{s}}$ encoded into bytes. The key calculation is also illustrated by figure 5.2.

Listing 5.2: Key generation process

```
8   for i := 0; i < K; i++ {
9     t_hat[i] = mulPolyVec(A_hat[i], s_hat)
10  }
11  t_hat = addPolyVec(t_hat, e_hat)
12  sk = encodePolyVec(s_hat)
13  pk = encodePolyVec(t_hat)
14  pk = append(pk, rho...)
```

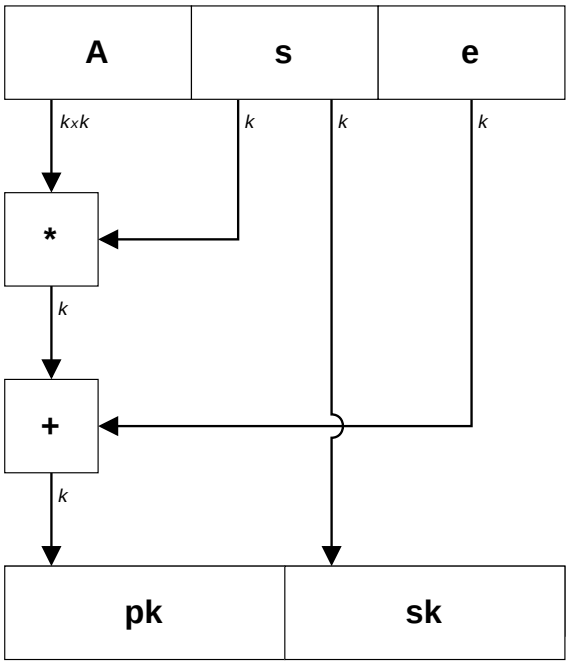


Fig. 5.2: Key calculation for `cpapkeKeyGen()`

The key generation functions is then encapsulated by another key generating function called `ccakemKeyGen()` in listing 5.3. The public key doesn't change but the secret key now also consists of the public key, 32 B hash of the public key and 32 B of randomness.

Listing 5.3: CCAKEM.KeyGen

```
60  pk, sk := cpapkeKeyGen()
61  sk = append(sk, pk)
62  sk = append(sk, hash32(pk))
63  sk = append(sk, randBytes(32))
```

5.3.4 Encapsulation

The encapsulation process relies on the encryption function – `cpapkeEnc()` – to work which which is illustrated in figure 5.3 and explained by by further text together with code snipets in listings. Firstly the parameters have to be setup as seen in listing 5.4. The public key (pk) parameter is decoded into $\hat{t}$ ρ is extracted and the matrix $\hat{A}$ is generated. A random polynomial vector $\hat{r}$ 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.

Listing 5.4: Encryption variable setup

```
21  A_hat := genPolyMat(rho)
22  r_hat := nttPolyVec(randPolyVec(r))
23  e1 := randPolyVec(r)
24  e2 := randPoly(r)
```

The parameter `u` is calculated by multiplying $\hat{A}$ and $\hat{r}$ plus the vector e_1 is added to it. Afterwards it is transformed from the NTT domain since booth factors are in NTT domain. The message `m` to be encrypted is decoded in order to create a polynomial from it and decompressed. The polynomial `v` is the factor of $\hat{t}$ and $\hat{r}$ and is again similarly transformed from the NTT domain. Then the polynomial e_1 is added to it together with the decoded message. Theses process can be seen in listing 5.5

As the last step booth `u` and `v` are firstly compressed and encoded to get them ready for network transfer (see listing 5.6). The parameters used in the compression and encoding processes are d_u and d_v reference in table 5.1

Listing 5.5: Encryption process

```
26  for i := 0; i < K; i++ {
27    u[i] = mulPolyVec(A_hat[i], r_hat)
28  }
29  u = addPolyVec(invNttPolyVec(u), e1)
30  parsed_m := decompress(decode(m))
31  v := invNtt(mulPolyVec(t_hat, r_hat))
32  v = polyAdd(v, e2)
33  v = polyAdd(v, parsed_m)
```

Listing 5.6: Creating ciphertext

```
35  for i := 0; i < K; i++ {
36    c1 = append(c1, encode(compress(u[i], Du), Du))
37  }
38  c2 := encode(compress(v, Dv), Dv)
39  c = append(c1, c2)
```

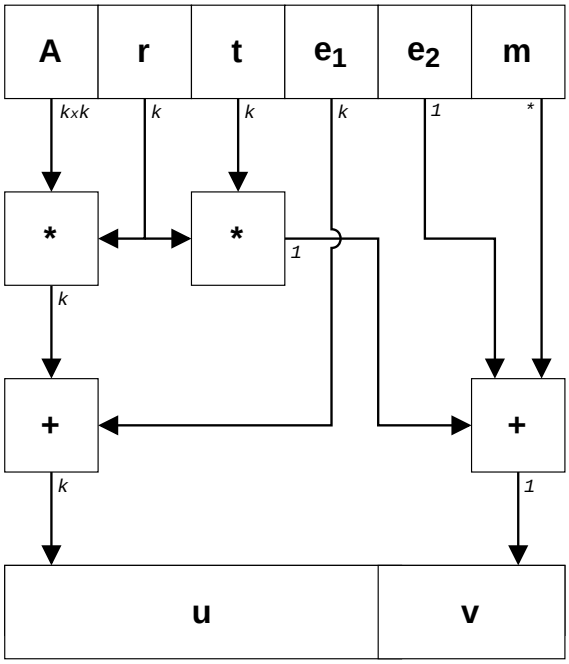


Fig. 5.3: Encryption for `cpapkeEnc()`

After the encryption function defined the encapsulation process (listing 5.7) can begin. Firstly a random message `m` is generated and hashed, then together with the hashed public key parameters K' and r are created using a hash function. Each parameter is the size of 32 B. The randomly generated message is generated and encrypted using the public key and random 32 bytes. Finally a key is derived using a KDF (Key Derivation Fucntion) with the inputs of K' and a hash of `m`. Booth the ciphertext and the derived key are then returned from the function.

Listing 5.7: CCAKEM.Enc

```
68  m := hash32(randBytes(32))
69  g_input = append(g_input, m)
70  g_input = append(g_input, hash32(pk))
71  K_dash, r := hash64(g_input)
72  c = cpapkeEnc(pk, m, r)
73  kdf_input = append(kdf_input, K_dash)
74  kdf_input = append(kdf_input, hash32(c))
75  key = kdf(kdf_input, 32) // Output 32 bytes
```

5.3.5 Decapsulation

Similarly as with the encapsulation process the decapsulation process requires the decryption function to be defined. The decryption process is also illustrated with a figure, specifically 5.4 and described in the following code snipets. Decryption process begins with decoding the parameters `u` and `v` from the ciphertext (see listing 5.8). Additionally the parameter `u` is transformed into NTT domain. The secret key used for decryption is also decoded.

Listing 5.8: Decryption variable setup

```
44  u_decoded := decodePolyVec(c, Du)
45  for i := 0; i < K; i++ {
46    u_hat[i] = ntt(decompress(u_decoded[i], Du))
47  }
48  c2 := c[POLY_BYTES*K:] // Get last POLY_BYTES bytes
49  v := decompress(decode(c2, Dv), Dv)
50  s_hat := decodePolyVec(sk, 12)
```

The actually decryption begins by multiplying $\hat{s}$ and $\hat{u}$ and transforming the product from NTT domain. It is then subtracted from `v` compressed and decoded to get the original message `m`. This process can be seen in listing 5.9

Listing 5.9: Decryption process

```
52  s_hat_u_hat := mulPolyVec(s_hat, u_hat)
53  invNtt(s_hat_u_hat)
54  m_decoded := polySub(v, s_hat_u_hat)
55  m = encode(compress(m_decoded, 1), 1)
```

Decapsulation starts by parsing the secret key. The randomly generated message generated in 5.7 is decrypted using the secret key. The variable r' is created using the same process as in 5.7 lines 69-71. Then another hashed ciphertext is created from m' and r'.

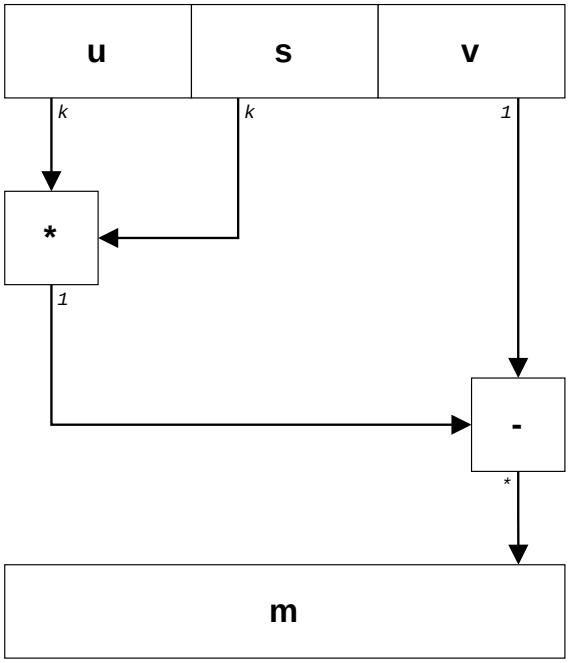


Fig. 5.4: Decription for `cpapkeDec()`

Listing 5.10: CCAKEM.Dec decapsulation part 1

```
85  m_dash := cpapkeDec(sk, c)
86  c_dash := cpapkeEnc(pk, m_dash, r_dash)
87  hash_c := hash32(c)
```

The code in the last listing in this section (5.11) derives the actual shared key. Though firstly an if statement has to be used. If the original ciphertext c and ciphertext created here c' are equal the value K' is used for generating the key, if they don't equal `z` is used. Additionally the hash of c' used to derive the key via a KDF.

Listing 5.11: CCAKEM.Dec decapsulation part 2

```
89  if BytesEqual(c, c_dash) {
90    kdf_input = append(kdf_input, k_dash)
91  } else {
92    kdf_input = append(kdf_input, z)
93  }
94  kdf_input = append(kdf_input, hash_c)
95  key = kdf(kdf_input, SharedKeySize)
```

5.4 CRYSTALS-Dilithium

Another algorithm from the group of lattice based cryptography is CRYSTALS-Dilithium signature scheme. It was also standardized during the 3rd round of 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 secret key without actually revealing it. Similarly to Kyber it also based on the MLWE problem. Dilithium also has a binding property which 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 unforgeble 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]

5.5 Implementing Dilithium

As with Kyber, Dilithium is implemented using only the standard go libraries and one external library [26] that contains implementations for `SHAKE-128` and `SHAKE-256` hash functions. Dilithium can be implemented in two ways, the first one by using a bigger public key. This implementation of Dilithium is also simpler overall. The other option is implementing a more complex algorithm which has a smaller public key by a factor of more then half. For this thesis the more complex implementation was chosen. The algorithm is explained at a very basic level in figure A.2. The small letter at the beginning of arrows denotes the number of polynomials it consists of. Firstly the public/secret keys are generated then the secret key is used in the signing process. A signature is generated which then can be verified by anyone who owns the related public key. Following sections explain all of these steps in more detail with the help code snippets. As before with Kyber these code snippets can't be actually compiled and are only a Go styled pseudocode. For the compilable implementation check the practical part of this thesis.

Tab. 5.2: Dilithium security levels [29]

	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 5.2 shows the individual parameters for each of the Dilithium parameter sets. The implementation in this thesis contains only Dilithium 2. When a param-

eter is relevant for the process being explained it will be mentioned and explained in that scenario instead of all the parameters being explained in this section.

5.5.1 Bit manipulation

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

$$\texttt{Decompose}(5687946,\ 1735) = 3278 * 1735 + 616. \tag{5.4}$$

As can be see in equation 5.4 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

$$\texttt{Power2Round}(5687946,\ 13) = 694 * 8192 + 2698, \tag{5.5}$$

$$\texttt{Decompose}(5687946,\ 8192) = 694 * 8192 + 2698. \tag{5.6}$$

Functions `MakeHint` and `UseHint` make use of the aforementioned functions to create and consume hints. The implementation for `MakeHint` is relatively simple and can be seen in listing 5.12. It returns true if the number $z + r_0$ is bigger then α because r_1 would either increase or decrease at least by one if that was the case. If the number z doesn't change the high bits of r it returns false and means that z doesn't affect the high bits of r.

Listing 5.12: `MakeHint` implementation

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

`UseHint` takes h, r and α as parameters. It can be used to calculate the high bits of $r + z$ without the knowledge of z and using only the hint h as seen in equation 5.7. The function first decomposes r and if the hint is true it will either add or subtract 1 from r_1 (high bits of r) depending on the sign of r_0.

$$\texttt{UseHint}(\texttt{MakeHint}(z,\ r,\ \alpha),\ r,\ \alpha) = \texttt{HighBits}(r + z,\ \alpha) \tag{5.7}$$

5.5.2 Theoretical basics and bit packing

Dilithium uses the same theoretical background as Kyber (see section 5.3.1) which includes rings, NTT transformation, polynomials and even uses the same n as can be seen in table 5.2.

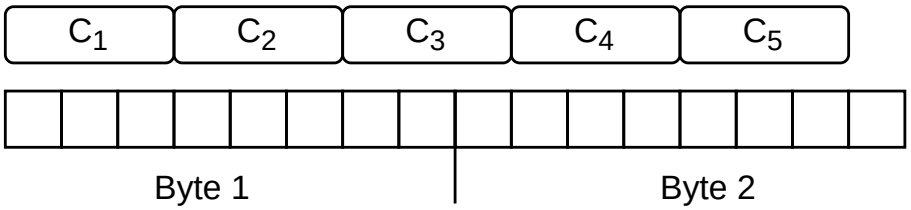


Fig. 5.5: Bit packing for vectors s_1 and s_2

Since Dilithium needs to transfer polynomials over the network whose coefficients are different sizes form each other a very efficient bit packing method can be used. For example the polynomial vectors s_1 and s_2 are in an interval $\{-2, -1, 0, 1, 2\}$ while using parameters for Dilithium 2. This means only 3 bits are required to pack a single coefficient into bit form (illustrated in figure 5.5). Although the coefficient firstly needs 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 in Dilithium 2 takes up 384 B. 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.

5.5.3 Key generation

Dilithium key generation process starts by creating a random seed and then using this seed to create another three seeds ρ, ρ', K. In listing 5.13 ρ is used to create the matrix $\hat{\mathrm{A}}$ and ρ' for generating error vectors s_1, s_2. Variable t is the product of $\hat{\mathrm{A}}$ and s_1 to which s_2 is added. t is used as the argument for `power2Round` (refer to subsection 5.5.1) function together with d which is a parameter defined 5.2. This functions splits t into t_1 and t_0.

Listing 5.13: Key generation

```
9   A_hat := expandA(rho)
10  s_1, s_2 := expandS(rho_dash)
11  for i := 0; i < L; i++ {
12    t[i] = mulPolyVec(A_hat[i], nttPolyVec(s_1))
13  }
14  t = addPolyVec(invNttPolyVec(t), s_2)
15  t_1, t_0 := powerToRoundPolyVec(t, D)
```

Variable t_1 is used as the public key together with the randomly generated K. The secret key consists of a hash of the public key, secret parameters s_1, s_2 and t_0. Polynomials vectors are additionally packed into bytes for easy transfer over the network as seen in listing 5.14. This process but simplified is also illustrated in figure 5.6.

Listing 5.14: Key construction

```
17  pk = append(rho, bitPack(t_1, 10))
18  sk = append(rho, K)
19  sk = append(sk, shake256(pk, 32))
20  sk = append(sk, bitPack(s_1, 3))
21  sk = append(sk, bitPack(s_2, 3))
22  sk = append(sk, bitPack(t_0, 13))
```

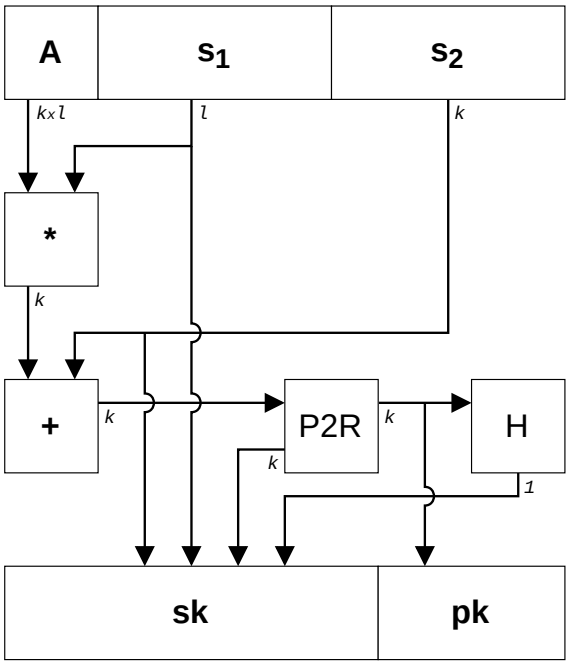


Fig. 5.6: Dilithium key generation

5.5.4 Signature creation

At the beginning of the Dilithium signing process (simplified illustration in figure 5.7) the secret key has to be parsed into variables which it consists off. This is done by unpacking the bytes into useful data. The NTT versions of s_1, s_2, t_0 can be precomputed ahead of time to increase the speed of singing.

Listing 5.15: Vectors generation

```
48  y := expandMask(rho_dash, kappa)
49  y_hat := nttPolyVec(y)
50  for i := 0; i < L; i++ {
51    w[i] = invNtt(mulPolyVec(A_hat[i], y_hat))
52  }
53  w_1 := highBitsPolyVec(w, 2*GammaTwo)
```

The message to be signed is hashed together with the hash of the public key tr and used later in the algorithm for generating the vector y and random bytes c. As seen in listing 5.15 the mentioned vector y is generated from ρ', it is then used to calculate another vector w. The `HighBits` function (see subsection 5.5.1) are then taken from every coefficient of every polynomial of vector w. The γ_2 is a parameter defined in 5.2.

The high bits of w are hashed together with the message hash to create a 32 B array of bytes c which is then used to scale vectors s_1 as seen in listing 5.16. The scaled vectors is added to the vector y which is the second part of the signature followed by the array of bytes c.

Listing 5.16: Signature generation

```
60  cs_1 := invNttPolyVec(scalePolyVecByPoly(s_1_hat, c_hat))
61  z := addPolyVec(y, cs_1),
```

The vector z together with c will be used to calculate the high bits of w by the verifier without the knowledge of y. For that to be feasible hints have to be created, this process is depicted in listing 5.17. The first step is to again scale the vector t_0 with c and invert the sign of all coefficients. The vector $w - c * s_2$ was calculated before hand and by by adding the scaled vector ct_0 to it is used to create the hints together with the inverted ct_0 (see `MakeHint` function in 5.5.1). The hits are then a part of the signature.

Listing 5.17: Hint generation

```
69  ct_0 := invNttPolyVec(scalePolyVecByPoly(t_0_hat, c_hat))
70  ct_0_inv := inversePolyVec(ct_0)
71  added_vecs := addPolyVec(w_sub_cs_2, ct_0)
72  h := makeHintPolyVec(ct_0_inv, added_vecs, GammaTwo*2)
```

The variables z, ct_0 and low bits of $w - cs_2$ (cs_2 is s_2 scaled by c) are also checked whether they reveal any part of the secret variables s_1, s_2 and t_0 by using a norm function. This function is simply returns the biggest absolute value of a polynomial coefficient in the whole vector. The norms of vectors are compared to either γ_1 or γ_2. If only one of the check fails the whole signing process is restarted with the only change to an integrable variable κ that makes sure the vector y is generated differently then last time. Another check that has to be made is for the number of hints. It can't exceed γ defined in 5.2.

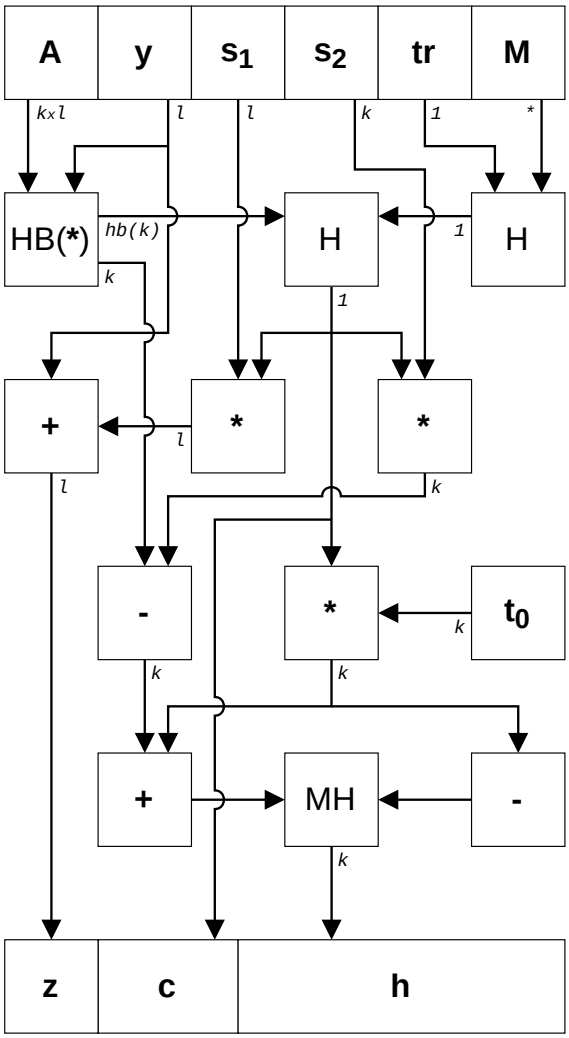


Fig. 5.7: Dilithium signature creation

5.5.5 Signature verification

The verification process for Dilithium starts off by unpacking required variables from the public key and the signature. $\hat{A}$ is generated the same way as in 5.13 line 9, it is then multiplied by the vector z. The public parameter t_1 is then at first scaled by 2^d to make up for the lower bits taken out by the `power2Round` function. It is then again scaled by c which is parsed from the signature.

Listing 5.18: Variable preperation

```
117   for i := 0; i < L; i++ {
118     Az[i] = mulPolyVec(A_hat[i], z_hat)
119   }
120   t_1_scaled_hat := nttPolyVec(scalePolyVecByInt(t_1, 1<<D))
121   ct_1 := scalePolyVecByPoly(t_1_scaled_hat, c_hat)
```

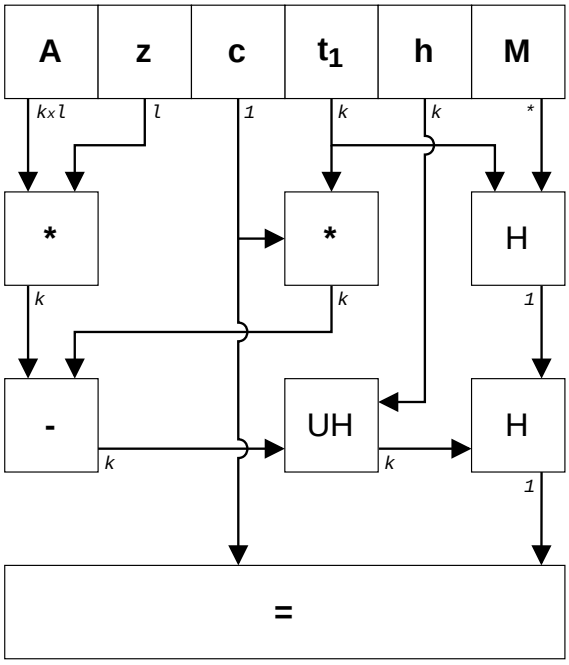


Fig. 5.8: Dilithium signature verification

The variable r is the result of subtracting ct_1 from Az. As mentioned in 5.5.4 the goal is to calculate the high bits of w. To achieve this the created hints are used on the vector r to create the exact copy of w_1. The hash of w_1 together with the message and some other shared parameters are then compared to c. If they are equal the verification process succeeded, if they don't equal the verification failed. See figure 5.8 for the simplified summary of the process.

Listing 5.19: Signature verification

```
123   r := invNttPolyVec(subPolyVec(Az, ct_1))
124   w_1 := useHintPolyVec(h, r, 2*GammaTwo)
125   shake = append(mi, bitUnpack(w_1, 6))
126   verified = BytesEqual(c, shake256(shake, 32))
```

5.6 Implementation results

The benchmarked values available in tables 5.3 and 5.4 were measured using a go program available in the practical part of this thesis, more specifically the code inside the `benchmark.go` file. One iteration of the benchmarking process for Kyber consists of key generation, key encapsulation and decapuslation. Similarly an iteration for Dilithium consists of key generation, signing process and a verification process. All benchmarks were done on AMD Ryzen 3600 cpu with a base core clock of 3.6 GHz. Implementations used to compare the performance to the thesis are the following:

- kyber-k2so by symbolicsoft
  - `https://github.com/SymbolicSoft/kyber-k2so`
- circl by cloudflare
  - `https://github.com/cloudflare/circl`

As is clear from the results of benchmarking, this implementation of the post quantum algorithms is substantially worse then other implementations. This is because the main goal behind the implementations was to get an understanding of how these algorithms work and so they are not optimized in any way other then the NTT transformation used for polynomial multiplication. One of the main goals for the second part of this diploma thesis ill be optimizing these algorithms.

Tab. 5.3: Kyber performace comparison

Implementation	5000 iterations	Average
this thesis	21.051 s	4210 µs
kyber-k2so	1.297 s	259 µs
circl	0.307 s	61 µs

Tab. 5.4: Dilithium performace comparison

Implementation	5000 iterations	Average
this thesis	19.868 s	3973 µs
circl	1.693 s	338 µs

Along with a guide on how to run/build the benchmarking application in appendix B it also contains a guide on how to test the correctness of Kyber and Dilithium implementations. In the case of Kyber the test only checks if the shared key is the same before and after decryption. The test for Dilithium checks if the signature is verified. These tests can be found in `main_test.go`.

Conclusion

The aim of this this was to introduce the reader with the possibility of a quantum supremacy where powerful enough quantum computers are capable of breaking modern cryptography. As was shown this might be a real possibility in the near future since IBM and other companies have started researching quantum computers and even building them. However the best quantum computer at the time of writing this is a 127 qubit one. Although that doesn't change the fact that post quantum algorithms which are resistent to quantum algorithm attack for public key cryptography are needed.

So far this thesis has introduced Kyber and Dilithium which are lattice-based post quantum algorithms standardized by NIST (National Insititue of Standards and Technology). The continuation of this thesis will include implementations for additional post quantum algorithms such as SPHINCS+ which is based on hashes and McEliece based on codes.

The implemented algorithms in this thesis are working correctly but are much less performant then other implementations such as the cloudflares circl library. One of the goals for the continuation of this thesis will be improving the performance of Kyber and Dilithium. Another goal will be creating a client-server communication protocol for exchanging data. The protocol will use the implementation of post quantum public key algorithms described in this thesis.

Bibliography

[1] BERNSTEIN, Daniel J. a Tanja LANGE. Post-quantum cryptography. *Nature* [online]. 2017, 14.9, **2017**(549), 188-194 [cit. 2022-10-09]. Dostupné z: 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. Dostupné z: 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 a 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 a Nicky MOUHA. *Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher*. 2nd ed. NIST Pubs, 2017, 32 s. Dostupné také z: 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 a Daniel SMITH-TONE. NISTIR 8105. *Report on Post-Quantum Cryptography*. NIST, 2016, 15 s. Dostupné také z: http://dx.doi.org/10.6028/NIST.IR.8105

[8] FIPS PUB 180-4. *Secure Hash Standard*. Gaithersburg, USA: NIST, 2015, 36 s. Dostupné také z: http://dx.doi.org/10.6028/NIST.FIPS.180-4

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

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

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

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

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

[14] PRETSON, Richard. *Applying Grover-s Algorithm to Hash Functions: A Software Perspective*. Bedford: The MITRE Corporation, 2022. Dostupné také z: 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. Dostupné také z: https://eprint.iacr.org/2015/1075

[16] IBM Unveils Breakthrough 127-Qubit Quantum Processor. IBM. *IBM Newsroom* [online]. 2021 [cit. 2022-10-26]. Dostupné z: https://newsroom.ibm.com/2021-11-16-IBM-Unveils-Breakthrough-127-Qubit-Quantum-Processor

[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]. Dostupné z: 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. Dostupné také z: 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]. Dostupné z: doi:https://doi.org/10.1145/237814.237838

[20] GOLDREICH, Obed, Shafi GOLDWASSER a 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]. Dostupné z: 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]. Dostupné z: 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] FOROUZAN, Behrouz. *TCP/IP Protocol Suite*. 4th edition. Raghothaman Srinivasan: McGraw-Hill, 2010, 1029 s. ISBN 978-0-07-337604-2.

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

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

[26] Crypto module. *Go Packages* [online]. [cit. 2022-11-17]. Dostupné z: `https://pkg.go.dev/golang.org/x/crypto`

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

[28] LIANG, Zhichuang, Shiyu SHEN, Yuantao SHI, Dongni SUN, Chongxuan ZHANG, Guoyun ZHANG, Yunlei ZHAO a 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]. Dostupné z: doi:10.1007/978-3-030-71852-7_28

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

Symbols and abbreviations

AES	Advanced Encryption Standard
CBD	Centrail Bionimal Distribution
CFB	Cipher Feedback
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
ECB	Electronic Code Book
ECC	Elitic Cruve Cryptography
ECDH	Elitic Curve Diffie-Hellman
ECDSA	Elitpic curve Digital Signature Algorithm
GCD	Greatest Common Divisor
GCM	Galois/Counter Mode
IFP	Integer Factorization Problem
KDF	Key Derivation Fucntion
KEM	Key Encapsulating Mechanism
KEP	Key Exchange Protocol
LWE	Learning With Errors
MAC	Message Authentication Code
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
UDP	User Datagram Protocol

List of appendices

A Lattice based algorithms

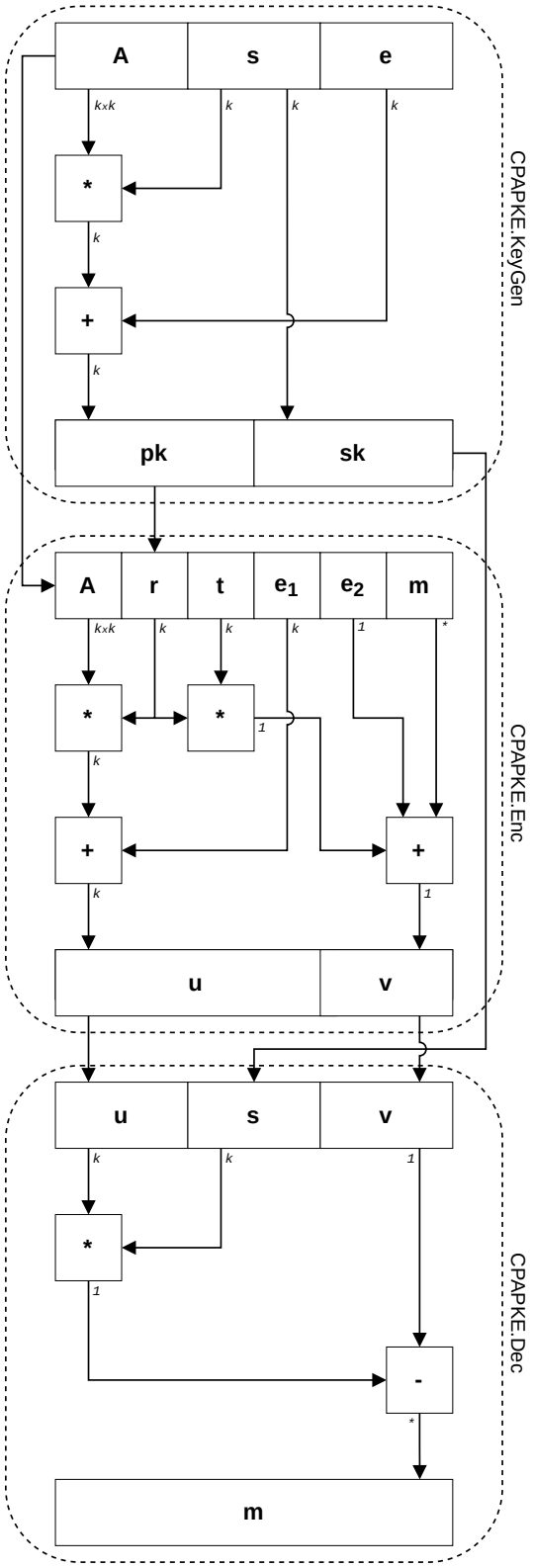


Fig. A.1: Kyber block scheme

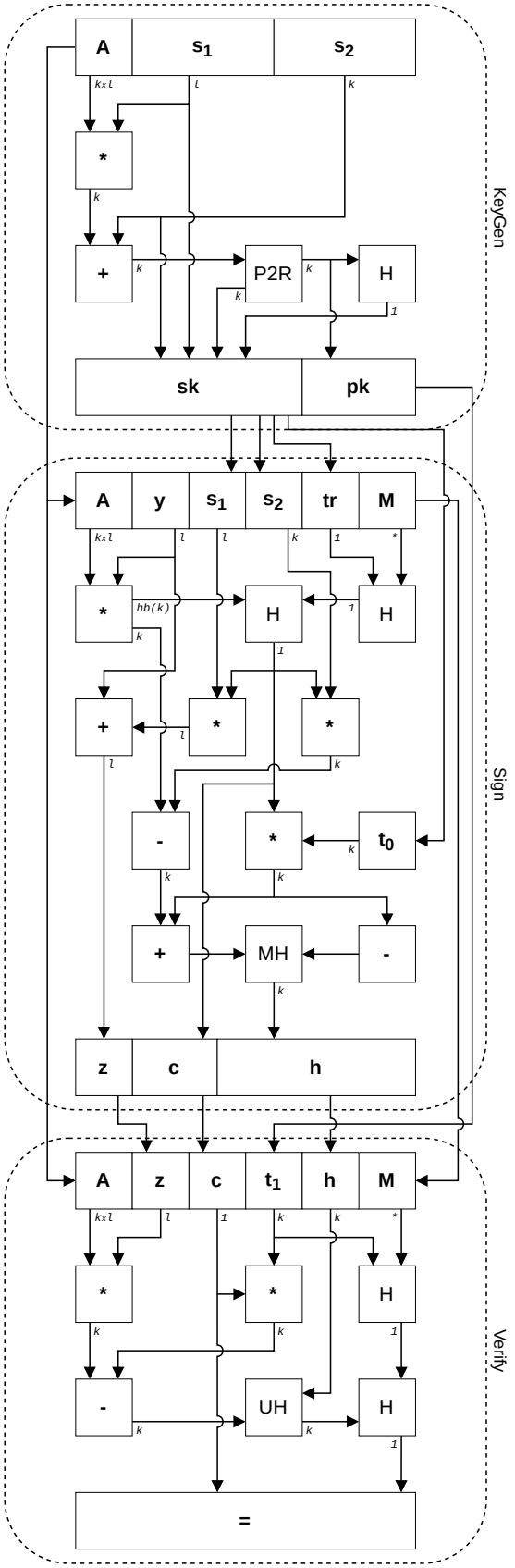


Fig. A.2: Dilithium block scheme

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 know operating systems such as Linux, Windows and Mac. First of all download the latest version of the go binary from this link

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

Once go is installed the binary for this thesis can be built by running

- `go build -v`

inside the command line interface of your operating system. The command also has to be ran inside the root directory of the project. The output of this command should yield a file named `main.exe` for Windows or `main` for other operating systems.

B.2 How to run

Once the binary is built it can be ran just like any other binary. For Linux run with

- `./main`

and for Windows run with

- `main`

in your command line interface. Rest of the examples will be applicable to the Linux operating system. Additionally the option `-i` can also be specified to change the number of iterations for the benchmark. For example to run benchmark with 5000 iterations run

- `./main -i 5000`

The default value for number of iterations is 1000.

B.3 How to test

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

- `go test -v`

into the command line interface in the root directory of the project.