

BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2021

Bc. Tatiana Novosadová



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

POST-QUANTUM CIPHERS

POSTKVANTOVÉ ŠIFRY

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR

AUTOR PRÁCE

Bc. Tatiana Novosadová

SUPERVISOR

VEDOUCÍ PRÁCE

doc. Ing. Jan Hajný, Ph.D.

BRNO 2021

Master's Thesis

Master's study program **Information Security**

Department of Telecommunications

Student: Bc. Tatiana Novosadová

ID: 195163

**Year of
study:** 2

Academic year: 2020/21

TITLE OF THESIS:

Post-Quantum Ciphers

INSTRUCTION:

The topic is focused on the analysis and development of an application for key establishment using modern cryptographic algorithms based on the principles of post-quantum cryptography. The result of the work will be realized as a software for key establishment using a protocol from NIST Round 3 competition [2]. The application will be delivered with complete documentation (user and installation manual), usable GUI and the report from the practical evaluation of basic parameters (such as time needed for key establishment, error rate, etc.). The software will be debugged and optimized.

RECOMMENDED LITERATURE:

[1] MENEZES, Alfred, Paul C. VAN OORSCHOT a Scott A. VANSTONE. Handbook of applied cryptography. Boca Raton: CRC Press, c1997. Discrete mathematics and its applications. ISBN 0-8493-8523-7.

[2] NIST: Post-Quantum Cryptography [online]. 2020 [cit. 2020-09-04]. Dostupné z:
<https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions>

**Date of project
specification:** 1.2.2021

Deadline for submission: 24.5.2021

Supervisor: doc. Ing. Jan Hajný, Ph.D.

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

WARNING:

The author of the Master's Thesis claims that by creating this thesis he/she did not infringe the rights of third persons and the personal and/or property rights of third persons were not subjected to derogatory treatment. The author is fully aware of the legal consequences of an infringement of provisions as per Section 11 and following of Act No 121/2000 Coll. on copyright and rights related to copyright and on amendments to some other laws (the Copyright Act) in the wording of subsequent directives including the possible criminal consequences as resulting from provisions of Part 2, Chapter VI, Article 4 of Criminal Code 40/2009 Coll.

ABSTRACT

The National Institute for Standards and Technology (NIST) has initiated a process to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptography algorithms through a public competition. An objective of this thesis is to study the available post-quantum algorithms for key establishment, that were published in the third round of this competition. After a proper analysis and comparison, one of the studied algorithms was implemented using available libraries for the chosen algorithm, the created program was optimized and documented.

KEYWORDS

Post-quantum cryptography, key-establishment, NIST, standardization, McEliece, NTRU, CRYSTALS-KYBER, SABER, lattice-based cryptography, MLWE

ABSTRAKT

Národní inštitút pre štandardy a technológie (NIST) zahájil proces na získanie, vyhodnotenie a štandardizáciu jedného alebo viacerých kryptografických algoritmov využívajúcich verejný kľúč prostredníctvom verejnej súťaže. Cieľom tejto diplomovej práce je nastudovať dostupné postkvantové algoritmy pre ustanovenie kľúča, ktoré boli zverejnené v treťom kole tejto súťaže. Po dôkladnej analýze a porovnaní bol jeden zo študovaných algoritmov implementovaný s využitím knižníc dostupných pre daný algoritmus, následne bol program optimalizovaný a zdokumentovaný.

KĽÚČOVÉ SLOVÁ

Post-quantum kryptografia, ustanovenie kľúčov, NIST, štandardizácia, McEliece, NTRU, CRYSTALS-KYBER, SABER, kryptografia založená na mriežkach, MLWE

NOVOSADOVÁ, Tatiana. *Post-quantum Ciphers*. Brno, 2021, 73 p. Master's Thesis. Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications. Advised by doc. Ing. Jan Hajný, Ph.D.

DECLARATION

I declare that I have written the Master's Thesis titled "Post-quantum Ciphers" independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this Master's Thesis, 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., Section 2, Head VI, Part 4.

Brno

.....

author's signature

ACKNOWLEDGEMENT

I would like to thank the advisor of my thesis, doc.Ing. Jan Hajný, Ph.D. for his valuable comments and quick replies to my questions and M.Sc. Sara Ricci Ph.D. for her guidance and help in understanding the theoretical background. I would also want to thank my family and Clemens Scholz for the support and encouragement.

Contents

Introduction	11
1 Quantum Computers	12
2 Post-Quantum Cryptography	14
2.1 Hash-Based Cryptography	15
2.2 Code-Based Cryptography	15
2.3 Lattice-Based Cryptography	16
2.4 Multivariate Cryptography	16
2.5 Supersingular Elliptic Curve Cryptography	17
3 NIST Standardization Competition	18
3.1 McEliece	18
3.2 NTRU	20
3.3 CRYSTALS-KYBER	23
3.4 SABER	26
3.5 Comparison of the Algorithms	29
3.5.1 Conclusion	31
3.5.2 The Best One Is...	33
4 Implementation of CRYSTALS-KYBER	34
4.1 Setting up the Environment	34
4.2 Exploring the Capabilities of the Code	35
4.3 Key Establishment	38
4.4 Time Needed to Establish the Keys	40
4.5 Suggested Adjustments	43
5 Adapted Implementation of CRYSTALS-KYBER	44
5.1 Changes and Specifications	44
5.2 Use of the New Implementation	48
5.3 Time Needed to Establish the Keys	50
Conclusion	52
Bibliography	54
List of Abbreviations	60
List of Appendices	62

A	Installation Guide	63
A.1	Application for macOS	63
A.2	Application for Ubuntu	64
A.3	Application for Other Operating Systems	65
B	User Manual	66
C	How to Create a New Executable	67
D	Shell Scripts	70
E	Contents of the Digital Attachment	73

List of Figures

3.1	Key agreement for NTRU.	22
3.2	Key agreement for Crystals-Kyber.	25
3.3	Key agreement for SABER.	28
4.1	Added information to CMakeLists.txt.	34
4.2	Automatically generated cmake build in CLion.	35
4.3	Output: Bytes sent during the key establishment.	36
4.4	Output: Generated key bytes.	37
4.5	Output: Speed of establishing the shared keys.	37
4.6	Output: Generated keys.	40
4.7	Output: Key establishment time (Test1).	42
4.8	Output: Key establishment time (Test2).	43
5.1	Key establishment process.	44
5.2	Flowchart for Alice's part of the key exchange.	46
5.3	Flowchart for Bob's part of the key exchange.	47
5.4	Terminal window server - input the parameters.	48
5.5	Terminal window client - input the parameters.	48
5.6	Terminal window server - server listening.	49
5.7	Terminal window client - start of the key generation.	49
5.8	Successfully calculated shared key.	49
5.9	Time needed to establish the keys for Alice and Bob separately.	50
A.1	Pop-up while opening the downloaded app on macOS.	63
A.2	Security & Privacy settings to allow the application to run.	64
B.1	Terminal prompt after trying to run the application.	66
C.1	Settings for the new project.	67
C.2	Contents of the attached project.	67
C.3	Pop-up question to confirm.	68
C.4	Executables directory.	69

List of Listings

4.1	Code: Parameters in params.h.	36
4.2	Code: Defined values of a seed.	36
4.3	Seed parameter for generating the matrix.	38
4.4	Called encapsulation function on Alice's side.	38
4.5	Called encapsulation function on Bob's side.	38
4.6	Called decapsulation function on Bob's side.	39
4.7	Called decapsulation function on Alice's side.	39
4.8	Called shake function for verification.	39
4.9	Key generation.	39
4.10	Added lines of code: Key establishment time.	41
A.1	Possible message after trying to run Crystals-Kyber.	64
A.2	Command to get the permission to run the application.	64
C.1	CMakeLists.txt to be edited.	68
C.2	Changes in CMakeLists.txt for Alice.	68
C.3	Level of security in params.h.	69
D.1	Shell script KyberServer.	70
D.2	Shell script KyberClient.	71

List of Tables

2.1	Security of cryptographic algorithms against quantum computers. . .	14
3.1	Comparison of the algorithms.	30
3.2	Concluded advantages and limitations of the algorithms.	32
5.1	The time needed to establish the keys based on the level of security (1).	51
5.2	The time needed to establish the keys based on the level of security (2).	51

Introduction

Cryptography in the present day is based on complicated mathematical problems, such as factorization or a discreet algorithm problem, which are all considered to be safe against known attacks performed on a standard computer. In the future however, in case of a successful production of a quantum computer using Shor's quantum algorithm for factorization, cryptography as we know it today would be defenceless as the mathematical problems would be solvable in a polynomial time. For this reason, post-quantum cryptography was created and now offers a variety of protocols and cryptosystems that should be safe against attacks performed on quantum computers. These new protocols are based on new mathematical problems, unsolvable in a polynomial time.

Even though a quantum computer functioning outside of laboratories in everyday conditions does not exist yet, the fast progressing development of technologies poses a potential threat to cryptography as we know it today, and peaks interest in post-quantum cryptography by many professionals.

The National Institute for Standards and Technology (NIST) has initiated a process to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptography algorithms through a public competition. As of July 22, 2020, candidates of the third round were published [1].

An objective of this thesis is to study the available post-quantum key-establishment algorithms published by NIST in the third round of the competition. After a proper analysis and comparison, one of the mentioned algorithms will be implemented using available libraries for the chosen algorithm.

1 Quantum Computers

Quantum computing makes use of the quantum phenomenon such as superposition of electrons to perform an enhanced computation. Superposition is a system that has two different states that define it and it can exist in both. An electron has two possible quantum states, known as spins: spin up and spin down. When an electron is in superposition, it is both up and down at once, it is a complex combination of both. Only after measuring it drops out of superposition and is observed to be in a specific spin state. In everyday life, this phenomenon can be compared to a coin tossing. While the coin is still in the air, it is both, heads and tails. Only after the coin falls down a specific result can be observed [2]. Computers that perform such computations are known as quantum computers and are believed to be able to solve computational problems like factorization faster than classical computers.

The first mentions about possible usage of a quantum phenomenon as a way to provide secure transmission of information date back to the sixties of the 20'th century. Only 20 years later Richard Feynman and Yuri Manin came up with an idea for a quantum computer [3].

A basic unit for a quantum computer is a quantum bit or a qubit. On the contrary of a bit, which is either 0 or 1, a qubit is a superposition of probabilities between 0 and 1, therefore its value lies on a spectrum and only by measuring, the value reaches 0 or 1. Two bits are required to describe a state of a single qubit, thus the amount of information contained in one qubit expands exponentially. For example, 10 qubits already contain 2^{10} bits of information. However, based on Holevo's theorem, also known as Holevo's bound, the amount of information we get after measuring n qubits, cannot exceed the n bits [4]. To use the advances of a quantum computer, certain algorithms need to be implemented to calculate the upper bound without having to measure it in the process. The most popular ones are Grover's algorithm [5] and Shor's algorithm [6].

The first demonstration of a quantum algorithm took place in 1998 when the *Nuclear Magnetic Resonance* (NMR) 2-qubit quantum computer was introduced and solved Deutch's problem [7]. In 2011 a machine produced by D-Wave Systems became the first-ever commercial quantum computer [8]. The company nowadays supplies *Nation Aeronautics and Space Administration* (NASA) with quantum computers using 2 000 qubits. In comparison, companies like IBM and Google use quantum computers with 50 - 72 qubits. The reason for the difference in the number of qubits is the technology each one of them uses (D-Wave uses so-called quantum annealing – a way of using quantum physics to solve optimization problems). Shor's algorithm cannot be running on a computer that uses this method. To be able to use all the possibilities of the quantum computer, the technology of the quantum

computer must be based on a quantum logic gate [9]. In 2018, Google announced the creation of a 72-qubit quantum chip called Birstlecon, but later it was proven to be difficult to control. In late September 2019, however, Google claimed to have reached quantum supremacy with a 53 qubit design called Sycamore. Quantum supremacy is a goal of demonstrating that a programmable quantum device can solve a problem that no classical computer is capable of solving in polynomial time. To reach quantum supremacy, a computer needs to consist of at least 50 qubits [10].

The latest progress in the field of quantum computing includes a creation of a silicone quantum processor that is able to function in a temperature of 1,5 Kelvin (many times warmer than common quantum processors) and a creation of a modification that allows quantum systems to stay operational (in other words coherent) for 10 000 times longer than before.

A group of researches in Sydney proved in April 2020 that a silicone-based quantum processor can operate in higher temperatures than other already existing processors. Even though this quantum processor was able to work in the temperature of 1,5 Kelvins, it is still far from daily conditions. The temperature needed for such processor is still very low, at $-271,65^{\circ}\text{C}$ after conversion from Kelvins [12].

In August 2020 a study titled *Universal coherence protection in a solid-state qubit* was released. A coherence equals to a lifetime of a single spin. The longer the lifetime of a spin, the more manipulations and quantum calculations can be performed, making the system more efficient. The usual way of keeping the system coherent is to physically isolate the system from the noisy surroundings, but this solution can be very complex and expensive due to an amount of materials needed. The study mentioned above proposes a new, more efficient way for reaching high coherence. Along with the usual electromagnetic pulses used to control quantum systems, an additional continuous alternating magnetic field is applied. When the field is precisely tuned, it is possible to rapidly rotate the electron spins, which allows the system to be unaffected by the rest of the noise. With this kind of a noise protection, the lifetime of a spin is 10 000 times longer than ever before [11].

Although there were some notable breakthroughs in the field of quantum computers in the past years, they are still far from being ready to be used in everyday life. Due to the necessity of cooling down to extreme temperatures, protection against magnetic fields and other external influences, the existing computers are for now unusable outside of laboratories.

2 Post-Quantum Cryptography

Cryptosystems used nowadays are mostly based on hard mathematical problems, i.e. integer factorization, discrete logarithm problem, Diffie-Hellman problem and an elliptic curve discrete logarithm problem (ECDLP). With a sufficiently powerful quantum computer using Shor's algorithm, the mentioned mathematical problems would be easily solvable in a polynomial time.

Tab. 2.1: Security of cryptographic algorithms against quantum computers.

Algorithm	Type	Purpose	Secure against quantum computers
AES-256	Symmetrical cryptosystem	Symmetrical encryption	Secure with longer keys
SHA-256, SHA-3	Hash function	One way compressing	Secure with longer outputs
RSA	Asymmetrical cryptosystem	Signing Asymmetrical encryption Key distribution	Not secure
ECDSA, ECDH	Asymmetrical cryptosystem	Signing Key distribution	Not secure
DSA	Asymmetrical cryptosystem	Signing Key distribution	Not secure

Table 2.1 shows the security of cryptographic algorithms with respect to the quantum computer attacks. Grover's algorithm [5] could offer a quadratic acceleration for quantum computers, making AES, SHA-2 and SHA-3 algorithms solvable even with larger key sizes or outputs. So far, cubic acceleration is impossible, therefore longer keys and outputs seem to be a sufficient solution in protection against quantum attacks. When it comes to ECDH, ECDSA, RSA, DSA and other cryptosystems based on similar mathematical problems that Shor's algorithm is able to solve, they cannot be considered safe [13].

The future vulnerability of currently popular algorithms has been brought to an attention at a *PQCrypto* conference in 2006 [14]. Since then, during a development of post-quantum ciphers, a great emphasis has been put on implementing different mathematical problems that would be capable of withstanding an attack by a quantum computer. More specifically problems, that are for now considered as unsolvable in a polynomial time. However, as mentioned in Chapter 1, a quantum computer powerful enough is still just a hypothetical machine as the existing experimental quantum computers do not possess the power needed to break an actual cryptographic algorithm. Thus, post-quantum cryptography currently consists of cryptographic algorithms that are only believed to be resistant to quantum computer attacks.

The post-quantum cryptography is divided into classes according to the problems each algorithm is based on [15].

2.1 Hash-Based Cryptography

A hash function is a one-way function that can be used to map data of any size to fixed-size data. It then creates a fingerprint of the input data, called a hash, for which it is very difficult to find the input data. A hash function $H : \{0, 1\}^* \rightarrow \{0, 1\}^s$ is cryptographically safe when it is preimage resistant, second preimage resistant and collision resistant [16].

- **Preimage resistance**

A hash function H is preimage resistant, if it is hard to find any m for a given h with $h = H(m)$.

- **Second preimage resistance**

A hash function H is second preimage resistant if it is hard to find any m_2 for a given m_1 with $H(m_1) = H(m_2)$.

- **Collision resistance**

A hash function H is collision resistant if it is hard to find a pair of m_1 and m_2 with $H(m_1) = H(m_2)$.

The security of hash functions might not be efficient enough for post-quantum cryptography when it comes to the collision resistance. In the post-quantum cryptography an attacker is not looking for same two values of a single hash, but for two same superpositions of a single hash.

In 2016, Professor Unruh introduced a reinforced security hash function, the so-called "collapsed" hash function. If the attacker knows the value of M , there are two possibilities for where this value came from. The system either determines the value M by measuring the superposition m , or the value M is estimated as $H(m)$. The attacker using the quantum computer does not know whether a value m or hash $H(m)$ is being used, therefore it will not be possible to find the original message [17].

2.2 Code-Based Cryptography

This category includes cryptographic algorithms, which functionality is based on error-correcting codes. An example of a code-based post-quantum algorithm is the McEliece cryptosystem, which uses Goppa codes. More about this cryptosystem is explained in chapter 3.1.

2.3 Lattice-Based Cryptography

Lattice cryptography is a general term for the construction of cryptographic algorithms used for key establishment, encryption and signing. Lattices are sets of points in a n -dimensional space arranged periodically. In case of 2-dimensional space, lattices could be described as a set of points in a field \mathbb{Z}^2 (\mathbb{Z}^2 is a cryptographic notation for a general field, denoted as \mathbb{K}^2)[18].

Definition 2.3.1. Assuming we have n linearly independent vectors $b_1, \dots, b_n \in \mathbb{R}^n$. A structure L is called a lattice over the vectors b_1, \dots, b_n , that could be mathematically described as:

$$L = \{a_1b_1 + \dots + a_nb_n \mid a_i \in \mathbb{Z}\}$$

Definition 2.3.2. If L is a lattice over vectors b_1, \dots, b_n , these said vectors form the base of the lattice L . The number n is a dimension of the lattice.

In cryptography, the dimension of the lattice n . Lattice-based cryptographic systems are considered resistant to both classical and quantum attacks, due to the use of mathematical problems that cannot be resolved effectively. These are the Shortest Vector Problem (SVP), the Closest Vector Problem (CVP) and the Shortest Independent Vectors Problem (SIVP) [18].

An example of a lattice-based cryptosystem that uses SVP is NTRU. More about NTRU and SVP is explained in Chapter 3.2. Another example of an algorithm based on lattices would be protocol Crystals-Kyber, that uses module learning with errors and is introduced in Chapter 3.3. Protocol Saber (Chapter 3.4) is also based on lattices and uses module learning with rounding problem.

2.4 Multivariate Cryptography

This type of post-quantum cryptography includes cryptographic algorithms based on multivariate polynomials over a finite field F . In case the polynomials are of second-degree, we are talking about multivariate quadratic cryptography. These cryptosystems use multivariate polynomial equations, of which the solution is proven to be nondeterministically polynomial-hard. Multivariate cryptography has proven to be the most successful in creating signature methods due to the fact that this type of post-quantum method provides the shortest signatures [19].

2.5 Supersingular Elliptic Curve Cryptography

An elliptic curve-based cryptography (ECC) is a public key method based on algebraic structures of elliptic curves over finite fields. Today's algorithms using elliptic curves (such as ECDH - Elliptic Curve Diffie-Helman) are not resistant to quantum attacks. The ECDH protocol uses points on a single curve, while supersingular curves are a group of at least five elliptic curves using unusually large endomorphism rings¹. Another difference is that private keys in post-quantum elliptic-curve cryptography are isogenies. Isogeny is a function that projects the points of one elliptic curve to another one, while preserving the vertices. The secret keys are the isogenies that lead from one elliptic curve to another. The public key is the supersingular elliptic curve itself [20].

In the next chapter we will introduce and analyze the finalists of the NIST standardization competition. As the protocols are part of some types of post-quantum cryptography mentioned above, we will describe a couple of the principles in greater detail on the following pages.

¹Endomorphism ring is a ring formed from an abelian group X using endomorphism. Endomorphism is a type of projection, a morphism from a mathematical object to itself.

3 NIST Standardization Competition

The National Institute for Standards and Technology (NIST) has initiated a process to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptography algorithms through a public competition in December 2016. As of July 22, 2020, NIST has begun the third round of public review. From the original 69 submissions are now only 15 eligible candidates for standardization. At the time of submitting this thesis, NIST has not made a decision on which of these algorithms will be added to national standards and considered officially quantum-safe [21].

In this thesis we will focus on the finalists in the field of post-quantum public-key encryption and key-establishment algorithms. In the following chapters we will analyze the McEliece, CRYSTALS-KYBER, NTRU and SABER post-quantum cryptosystems. These cryptosystems will be compared based on variety of parameters, such as computational complexity, availability of the source codes, hardware requirements, etc. After analyzing all the available data for comparison, we will be able to determine a possible best candidate for standardization.

3.1 McEliece

The McEliece cryptosystem is an asymmetric algorithm developed by Robert McEliece in 1978. It is one of the oldest cryptosystems with a public key, based on error-correcting codes, more specifically Goppa codes.

The basic general idea of the McEliece cryptosystem lies in syndrome decoding of linear error-correcting codes¹. If the number of errors is not bounded, the problem falls under the category of NP-complete problems. There are classes of linear codes with a very fast decoding algorithm. For this reason, McEliece takes one of these linear classes and disguises them. While the attacker is forced to use syndrome decoding to decrypt the shared information, the receiving side can remove the disguise and use the fast decoding algorithm. Any linear code with a good decoding algorithm can be used, McEliece suggested to use Goppa Codes [22].

McEliece uses a binary version of a Goppa code, which is an error-correcting code. The binary structure comes with several mathematical advantages over non-binary variants and provides a better use in computer technology and cryptography.

Definition 3.1.1. For each irreducible polynomial of degree t over $GF(2^m)$ exists a binary irreducible Goppa code of a length $n = 2^m$ and a dimension $k \geq n - tm$, capable of correcting any pattern for decoding these codes [23].

¹Syndrome decoding is a method of decoding a linear code over a noisy channel (one on which errors are made).

McEliece consists of three algorithms:

- a probabilistic key generation algorithm which produces a public and a private key,
- a probabilistic encryption algorithm,
- a deterministic decryption algorithm.

Key Generation

All participants in the communication use the same secured parameters - n , k , t [23]. Value n defines the length of the Goppa Code, as mentioned in the definition 3.1.1. Value t is a degree of an irreducible polynomial and a value k sets the dimension for the chosen matrices.

- A matrix G of a size $k \times n$ is created. This matrix is a generator for the code.
- A random binary non-singular matrix S of a size $k \times k$ is chosen.
- A random permutation P of a size $n \times n$ is chosen.
- From the generated matrices S, G and a permutation P a matrix G' is calculated; $G' = SGP$. The matrix G' is called a public generator matrix, since this value will be publicly known. G' is therefore the disguised matrix mentioned at the beginning of this chapter.
- **return** $P_k = (G', t)$.
- **return** $S_k = (S, G, P)$.

As mentioned before, the attacker has an access to the disguised matrix G' and in order to get the original G matrix he needs to use syndrome decoding. Since the number of errors used to disguise the matrix is not defined, the problem can be considered as unsolvable.

Key Agreement Protocol

The following process serves for a quantum key distribution (QKD). The QKD is not a part of an official submission for the NIST competition. As a result of this protocol, both parties should obtain the same shared key k [24].

- Alice sends a request to reserve a pair of QKD servers.
- Alice generates a random number m . This number should have the same length as the desired length of a shared key k . Value m gets encrypted using a part public key generated in the previous step and then sent to a her QKD server.

$$E = (G', m)$$

- Bob generates a random number m' that is of a same length as key k . Value m' gets encrypted using a part public key generated in the previous step and then sent to his QKD server.

$$E = (t, m')$$

- The QKD servers decrypt the values m and m' using Alice's and Bob's secret keys and generate the key k .
- The servers calculate:

$$x = k \oplus m$$

$$x' = k \oplus m'$$

- Alice receives the value x and recovers $k = x \oplus m$.
- Bob receives the value x' and recovers $k = x' \oplus m'$.

While in classic cryptography this algorithm did not gain much acceptance nor popularity, mostly due to the big size of the keys (for example for reaching an 80-bit security, a Goppa code-based cryptosystem needs 460 647-bit public keys [25]), it seems to be a perfect candidate for quantum cryptography, as it is immune to attacks using Shor's algorithm.

Even though the McEliece cryptosystem is only being considered for standardization in the field of key-establishment, it is also a suitable algorithm for encryption, decryption or digital signatures. As mentioned above, the only disadvantage of the algorithm is the need for longer keys in order to reach a desirable level of security against quantum computers.

3.2 NTRU

The NTRU protocol is a key encapsulation mechanism (KEM) scheme based on a shortest vector problem (SVP) and a Ring Learning with errors. The issue of a SVP is associated with a search for the shortest base vector. As an example, we can imagine a given lattice L with a random base, where the goal is to find the shortest non-zero vector that still belongs to the lattice (the lattice structure and its base are explained in chapter 2.3). We are therefore looking for a vector, that is close to zero, but its value still is not equal to zero [26, p. 370–372].

The Learning With Errors problem was not originally created for lattices. The SVP problem was proved to be equivalent to LWE, making the LWE applicable on lattices and making the LWE officially post-quantum proof (after proving the same hardness of the problem) [27, article 129].

The LWE problem works with arbitrary lattices, that were already introduced and described in chapter 2.3. Arbitrary lattices use real numbers in order to build the lattice structure. The LWE also uses an error distribution (usually a Gaussian error distribution with a relative error rate $\alpha < 1$) in order to make the solution of the problem harder.

Definition 3.2.1. Taking a n number of LWE samples, the associated vector $\mathbf{b} = b_1, \dots, b_n$ can be defined as ($\mathbf{b} = \mathbf{A}s + e$). \mathbf{A} is a random matrix in $\mathbb{Z}_q^{m \times n}$, s is a secret uniform random vector in \mathbb{Z}_q^n and e is a small noise value from a chosen distribution. The LWE problem is based on the fact that it is difficult to find values that solve ($\mathbf{b} = \mathbf{A}s + e$), even when the values for \mathbf{A} and \mathbf{b} are publicly known.

A RLWE uses ideal lattices instead of arbitrary ones.

Definition 3.2.2. Informal definition: An ideal lattice is a general name for any cyclic lattice. A cyclic lattice is a sublattice of a set \mathbb{Z}^n preserved with a rotational shift operator.

The specific type of an ideal lattice that is used for NTRU protocol is a polynomial ring, which is a cyclic structure built by a set of polynomials over a finite field \mathbb{Z} . A RLWE also uses an error distribution, in case of NTRU it is usually a discretized Gaussian distribution, with the same relative error rate as in the original LWE. The principle of the RLWE problem stays the same as described in a definition 3.2.1, with a difference of using polynomials as vector spaces, creating an n -dimensional ring, rather than just a single scalar as in LWE.

A use of an ideal lattice in a cryptosystem decreases the number of parameters necessary to describe a lattice by a square root, requiring less computational time and making the algorithm more efficient [28].

Key Generation

Keys generated by NTRU have a form a matrix, therefore for the generation three parameters are necessary - N, p, q .

- For p and q a rule must hold, where $\gcd(p, q) = 1 \cup q \gg p$.
- All the polynomials lie in the ring $R = \mathbb{Z}[X]/(X^N - 1)$.
- Polynomials f and g are chosen, so that the polynomial f has an inverse modulo to p and q .
- These inverse values are then denoted as F_p and F_q :

$$F_p \times f \equiv 1 \pmod{p}$$

$$F_q \times f \equiv 1 \pmod{q}$$

- The public key is the polynomial:

$$h \equiv f^{-1} \times g \pmod{q}$$

- Parameters N, p, q are public too.
- The private key is the polynomial f along with F_p .

The above mentioned way of establishing the keys is a part of the NTRU Encrypt protocol. NTRU can also be used for digital signatures with its NTRU Sign variant [29]. Even though the protocol was patented in the 90's, an open-source code is now publicly available on the official website of the submission and a linked GitHub repository [43].

Key Agreement Protocol

The Key Agreement Protocol for NTRU [30] is graphically shown and described below. This protocol uses the public key h created during the Key Generation in previous section.

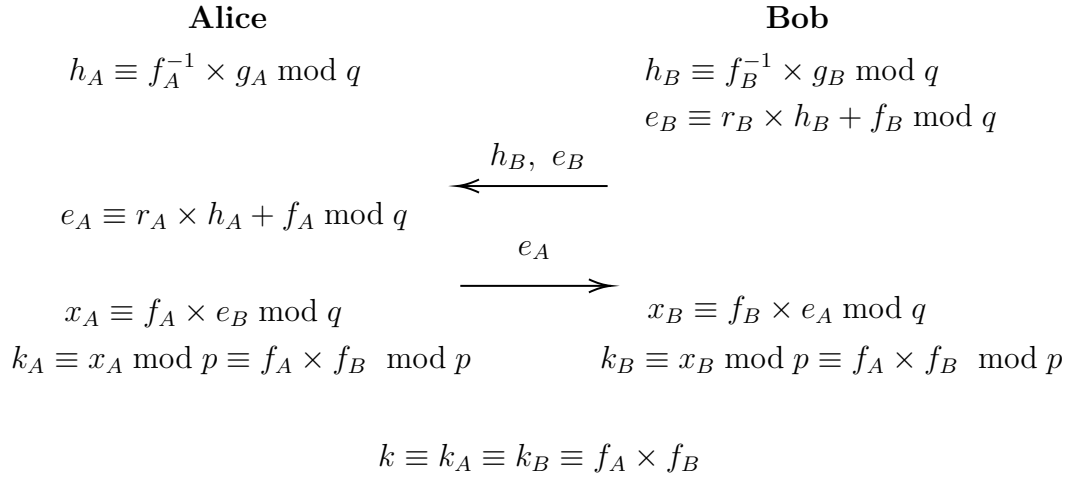


Fig. 3.1: Key agreement for NTRU.

The flow shown in the figure 3.1 is described in the following steps.

- Alice computes her public key h_A and sends it to Bob. As described in a previous section, the computation is:

$$h_A \equiv f_A^{-1} \times g_A \pmod{q}$$

- Bob computes his public key h_B and value e_B , where r_B is a randomly chosen polynomial with small coefficients:

$$h_B \equiv f_B^{-1} \times g_B \pmod{q}$$

$$e_B \equiv r_B \times h_b + f_B \pmod{q}$$

- Bob sends h_B and e_B to Alice.
- Alice chooses r_A and computes e_A :

$$e_A \equiv r_A \times h_A + f_A \pmod{q}$$

- Alice sends e_A to Bob and computes x_A and k_A :

$$x_A \equiv f_A \times e_B \pmod{q}$$

$$k_A \equiv x_A \pmod{p} \equiv f_A \times f_B \pmod{p}$$

- Bob computes x_A and k_B :

$$x_B \equiv f_B \times e_A \pmod{q}$$

$$k_B \equiv x_B \pmod{p} \equiv f_A \times f_B \pmod{p}$$

- A shared session key is k :

$$k \equiv k_A \equiv k_B \equiv f_a \times f_B \pmod{p}$$

3.3 CRYSTALS-KYBER

Protocol KYBER is one of the two cryptographic primitives contained in the Cryptographic Suite for Algebraic Lattices (CRYSTALS). Kyber is an IND-CCA2-secure² key encapsulation mechanism (KEM), which security is based on the hardness of solving the learning-with-errors problem over module lattices (MLWE).

The difference between a MLWE and RLWE (described in chapter 3.2), is that the MLWE uses module structures instead of rings.

Definition 3.3.1. A module is an algebraic structure generalizing rings and vector spaces, and module lattices generalize both arbitrary lattices and ideal lattices.

Definition 3.3.2. Informal definition: A Module Learning With Errors is the RLWE problem, where the elements of the ring are replaced by the elements of the module.

Most constructions based on RLWE can be adapted to MLWE, however with higher requirements on memory [31]. As MLWE is a version of LWE, a formal definition 3.2.1 is also applicable in this case.

²The scheme achieves the indistinguishability notion, even if an attacker has access to a public key and a decryption oracle.

Key Generation

The key generation algorithm KeyGen returns a pair (pk, sk) consisting of a public key and a secret key [32]. The parameter $n = 256$. Parameter k defines the number of dimensions for the used vectors. The key generation algorithm follows these steps:

- A public seed ρ is chosen from $\{0, 1\}^n$.
- A uniform matrix $\mathbf{A} \in R_q^{k \times k}$ is created, using the seed ρ , where R_q is a ring and $k \times k$ defines the size of the matrix \mathbf{A} over the ring R_q .
- Secret coefficients $(\mathbf{s}, \mathbf{e}) \in \beta_\eta^k$ are chosen, where β_η^k is a binomial distribution for a positive integer η .
- Value of \mathbf{b} is calculated: $\mathbf{b} = (\mathbf{A}\mathbf{s} + \mathbf{e}, d_t)$; d_t is a chosen positive integer parameter, a recommended value for d_t is $d_t = 10$ or $d_t = 11$.
- **return** $pk = \mathbf{b}, \rho$; $sk = \mathbf{s}$.

The algorithm in the NIST submission lists three different parameter sets aiming at different security levels. Specifically, Kyber-512 aims at security roughly equivalent to AES-128, Kyber-768 aims at security roughly equivalent to AES-192, and Kyber-1024 aims at security roughly equivalent to AES-256 [33].

Key Agreement Protocol

A key agreement is in case of Crystals-Kyber labeled as a key encapsulation. The result of this process is a shared key k . The process of establishing the key works with some of the values introduced in the key generation part above 3.3. The following steps describe in detail the key-establishment visually documented in Figure 3.2.

- Alice and Bob generate their public and secret keys as described above.
- Bob sends his pk_B to Alice.
- Alice chooses the value m from $\{0, 1\}^n$.
- Alice calculates cA . This value is calculated by using the hash functions. $G : \{0, 1\}^* \rightarrow \{0, 1\}^{2 \times n}$ and $H : \{0, 1\}^* \rightarrow \{0, 1\}^n$, Bob's public key pk_B and $m \in \mathcal{M}$; $\mathcal{M} = \{0, 1\}^n$:

$$cA = G(H(pk_B), m)$$

- Alice sends values pk_A and cA to Bob.
- Bob also chooses his value m from $\{0, 1\}^n$.
- To calculate the value of KB , Bob does the following:
 - Parameters d_u and d_v are chosen, both being positive integers.
 - Secret coefficients $(e_1, e_2) \in \beta_\eta^k$ are chosen, where β_η^k is a binomial distribution for a positive integer η .

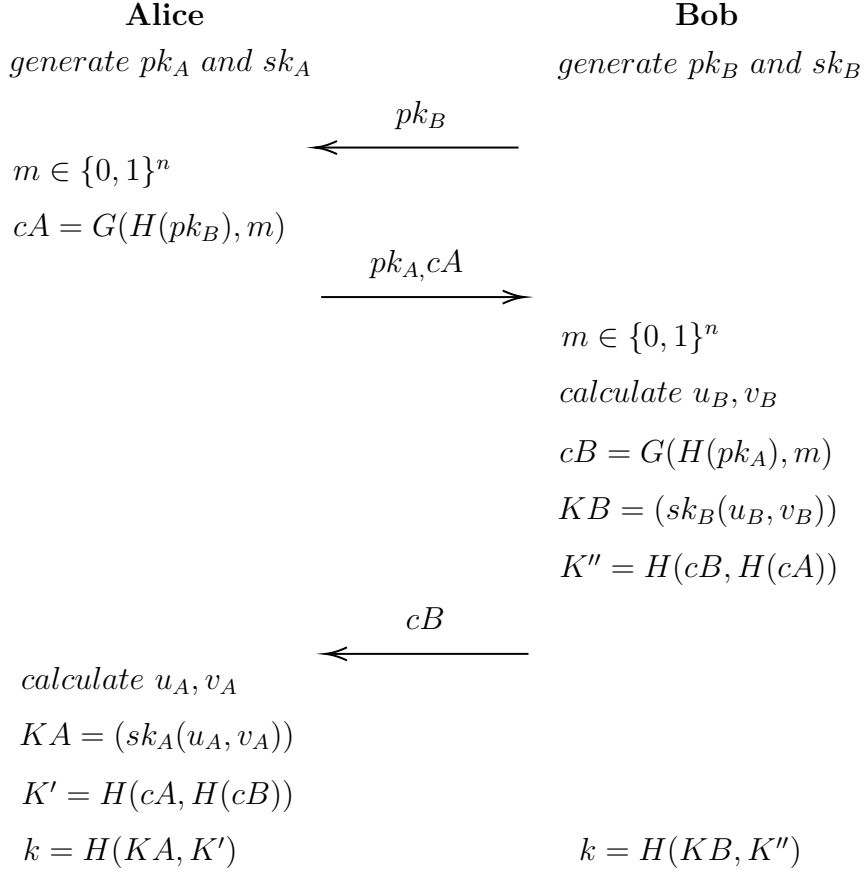


Fig. 3.2: Key agreement for Crystals-Kyber.

- A value u is calculated (A^T being a transpose of the matrix A created during the key generation):

$$u = (A^T r + e_1, d_u)$$

- A value v is calculated:

$$v = (t^T r + e_2 + \frac{q}{2} \times m, d_v)$$

- Using the values v_B and u_B calculated as described above, and his secret key, Bob then calculates KB :

$$KB = (sk_B(u_B, v_B))$$

- Bob calculates cB in a similar manner as Alice:

$$cB = GH(pk_A), m$$

- Bob calculates the value of $K'' = H(cB, H(cA))$.
- Bob sends the value of cB to Alice.

- Value KA is calculated using sk_A and u_A, v_A (values of u_A, v_A are obtained with the same calculations as performed by Bob):

$$KA = (sk_A(u_A, v_A))$$

- Alice calculates $K' = H(cA, H(cB))'$.
- Both sides now calculate the shared key. For Alice, the shared key $\mathbf{k} = H(KA, K')$ and for Bob the shared key $\mathbf{k} = H(KB, K'')$. These values are equal.

3.4 SABER

Protocol Saber is yet another protocol based on lattices that made it into the final round of the NIST standardization competition. SABER is an IND-CCA2 secure Key Encapsulation Mechanism, which security relies on the hardness of the Module Learning With Rounding problem (MLWR). Module structures are defined in the chapter 3.3.

The principle of LWR is also described by the definition 3.2.1. A small noise value e however is not determined by an error distribution. While the LWE problem adds a random small error to various samples $\langle a, s \rangle \in \mathbb{Z}_q$ to hide their exact value, the LWR uses a deterministically rounded version of $\langle a, s \rangle$. LWR can be seen as a de-randomized LWE, more closely defined by the definition 3.4.1.

Definition 3.4.1. For some $p < q$, the elements of \mathbb{Z}_q are divided into p contiguous intervals of roughly q/p number of elements each and define the rounding function $f_{round} : \mathbb{Z}_q \rightarrow \mathbb{Z}_p$. This function maps $x \in \mathbb{Z}_q$ into the index of interval that x belongs to. For example if q and p are both powers of 2, this could correspond to the $\log(p)$ of the most significant bits of x .

In simpler words: In LWR based schemes, the noise is deterministically obtained by scaling down from a modulus q to a modulus p . This naturally reduces the size of the public keys and ciphertexts and lowers the overall number of secret polynomials that need to be sampled. The use of LWR also reduces the amount of randomness required compared to LWE based schemes to a half, and with the smaller sizes of keys decreases the bandwidth [34, p.2-5],[35].

Key Generation

Parameters that are specified for Saber, including some recommendations to make the algorithm quantum-safe are:

- n, l : The degree of the polynomial ring $\mathbb{Z}_q[x]/(X^n + 1)$ is $n = 256$. The rank of the used module l determines the dimension of the lattice problem. A specific value for this dimension is not defined, the higher the dimension the higher security, this however might reduce correctness.
- R_q is a quotient of the ring $\mathbb{Z}_q[x]/(X^n + 1)$ and the value n
- q, p, T : The values of p, q, T are meant to be any power of 2. $q = 2^{\epsilon q}$; $p = 2^{\epsilon p}$; $T = 2^{\epsilon T}$, where $\epsilon q > \epsilon p > \epsilon T$. Not following this recommendation will result in lower security.
- μ : A parameter used to specify the coefficient of a secret vector in the lattice according to a binomial distribution; $\mu < p$.
- gen : An output function that is used to generate a pseudo-random matrix from a seed $seed_A$.
- NOTE: $R^{l \times l}$ denotes the ring, defined by the matrices of a size $l \times l$ over the ring R .

The Saber key generation is specified by the following process:

- A $seed_A$ is picked from $\{0, 1\}^n$.
- A pseudo-random matrix \mathbf{A} is generated using the seed chosen in the first step: $\mathbf{A} = gen(seed_A)$; $\mathbf{A} \in R_q^{l \times l}$.
- The secret vector \mathbf{s} , is sampled according to a binomial distribution $\beta_\mu(R_q^{l \times l})$: $\mathbf{s} = \beta_\mu(R_q^{l \times l})$.
- The value of \mathbf{b} is calculated using the generated matrix \mathbf{A} , the secret vector \mathbf{s} and a constant vector h used for rounding³: $\mathbf{b} = (\mathbf{A}^T \mathbf{s} + h)$.
- **return** $pk = (seed_A, \mathbf{b})$.
- **return** $sk = \mathbf{s}$.

Due to the use of powers of 2 in parameters p, q and T , the scaling and rounding operations are significantly simplified and thanks to the use of MLWR, the bandwidth necessary for the algorithm is notably lower than in other similar systems using MLWE (for example Crystals-Kyber, explained in chapter 3.3) [35].

Key Agreement Protocol

The key agreement protocol works with some of the values calculated during the key generation. In the figure 3.3 we can see the flow of establishing the key. In SABER scheme, according to the documentation the two communicating parties sometimes fail to agree on the same key. The probability of this failure can be made negligibly

³The constant vector is used to replace the rounding operations by a simple bit shift to the right. The bit shift however still mimics the rounding operation.

small by sending some additional reconciliation data c . The key exchange flow is described by the following steps [36]:

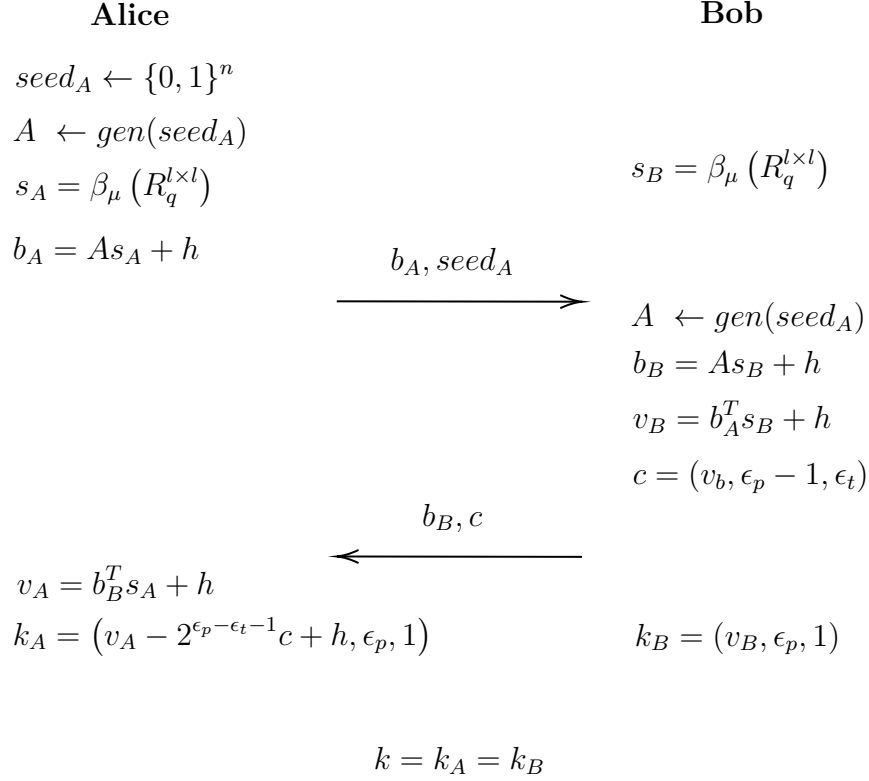


Fig. 3.3: Key agreement for SABER.

- Alice calculates A , s_A and b_A as described in previous section.
- Alice sends the value b_A and the $seed_A$ to Bob.
- Bob calculates the values s_B and A using the seed received from Alice. He then calculates b_B , v_B and c :

$$b_B = A^T s_B + h$$

$$v_B = b_A^T s_B + h$$

$$c = (v_B, \epsilon_p - 1, \epsilon_t)$$

- Bob sends the values b_B and c to Alice.
- Alice computes the value v_A :

$$v_A = b_B^T s_A + h$$

- Alice then computes the key k_A :

$$k_A = (v_A - 2^{\epsilon_p - \epsilon_t - 1} c + h, \epsilon_p, 1)$$

- Bob computes the key k_B .

$$k_B = (v_B, \epsilon_p, 1)$$

- The shared key $\mathbf{k} = k_A = k_B$.

3.5 Comparison of the Algorithms

After a technical introduction of all the finalists of the NIST competition in the key-establishment category, it is now possible to analyze and compare these algorithms.

The overview of similarities and differences of the algorithms is noted in the table 3.1. On the following pages we analyze the advantages and disadvantages of each of the algorithms.

As the cryptosystems NTRU, CRYSTALS-KYBER and SABER are all based on lattices and a learning problem, it is easier to compare the pros and cons of their mathematical problems. The LWE is a well studied and now understood problem, giving NTRU and CRYSTALS-KYBER an advantage in general knowledge of how the system works [33, p.28].

Module lattices have more complicated algebraic structures than ideal lattices. Thus, MLWE might be able to offer a better level of security than RLWE and still have performance advantages over plain LWE. However, this assumption is still only weakly supported, therefore it cannot be said for sure, whether MLWE is better than RLWE [38].

Thanks to the MLWR that is used for SABER, the band-width necessary for the algorithm is notably lower than in other similar systems using MLWE, which is used for Crystals-Kyber. This happens due to reducing the randomness that is required to a half [34, p.2-5].

The McEliece protocol very obviously requires much bigger key sizes than other protocols. While using Goppa codes, the recommended parameters are length $n = 6\,960$, dimension $k = 5\,413$ and number of errors $t = 119$. With parameters set like this, McEliece is expected to be secure against quantum attacks, however the size of the public key goes all the way up to 1 046 738,875 bytes [39]. McEliece key-generation software is not very fast. Even though none of the publications specify the exact time needed, it is mentioned in multiple documents, that due to the large length of the key, the time to generate this key is also longer. Therefore, applications must continue using each public key for long enough to handle the costs of generating and distributing the key [37].

Protocol CRYSTALS-KYBER uses the longest keys compared to other lattice-based cryptosystems [40]. The table includes the key sizes for Kyber-1 024, that aims at security roughly equivalent to AES-256. Despite the length of the keys, the

Tab. 3.1: Comparison of the algorithms.

Algorithm	Mathematical problem the algorithm is based on	Secret key size [bytes]	Public key size [bytes]	Time needed to establish keys [μ s]	Code available?	Language of the code	Software implementations	Hardware implementations
McEliece	Hardness of decoding a linear code (Goppa code)	?	1046738 [39]	?	yes [42]	C	SUPERCOP [42]	Artix-7 FPGA Virtex-7 FPGA [42]
NTRU	RLWE problem	1422 [44]	1140 [44]	?	yes [43]	Python C	Open Quantum Safe AVX2 [43]	?
CRYSTALS-KYBER	MLWE problem	3168 [40]	1568 [40]	1.5 K (ASIC) [40]	yes [40]	C	PQClean BoringSSL SUPERCOP [40]	ASIC [40]
SABER	MLWR problem	2304(1344) [41]	992 [41]	21.8 (UltraScale+) [45, p.20]	yes [41]	C	C AVX Cortex-M0 Cortex-M4 [41]	Artix-7 FPGA UltraScale+ FPGA [41]

algorithm is capable of establishing the keys in the shortest amount of time [45, p.20] using Application-specific integrated circuit (ASIC) hardware.

The sizes of keys for CRYSTALS-KYBER and SABER are taken from official websites of submissions to the NIST competition [40][41], just like the times needed to establish the keys. The information about NTRU key sizes are taken from the official algorithm specification documentation [44] and the size of a public key for McEliece was taken from the publication [39]. Some of the data needed for a proper comparison unfortunately was not available at any sources. When it comes to comparing the time necessary for establishing the keys, the times available from sources were not measured on the same device, making the comparison less accurate.

Considering that the next part of this thesis is an implementation of a chosen, most suitable algorithm for a post-quantum cryptography, it is important to evaluate whether the source codes are available, which language they are written in, and compare their software and hardware requirements. All the information relating to the code and implementations stated in the table was taken from the official websites of the protocols [40][41][42][43] and their linked GitHub repositories.

As all of the source codes are publicly available and mostly use C as its primary programming language, there is not much to compare. When it comes to deciding which hardware or software implementation will be used, it is a matter of personal preference, however this does not make any of the algorithms better than the other. Rather than a comparison, the "code" part of the table gives an overview of the options each algorithm has to offer when it comes to their implementation.

3.5.1 Conclusion

The table 3.2, contains conclusions of the advantages and limitations. The information in this table is based on the content of previous pages and extra supporting documentation.

The McEliece cryptosystem seems to have more limitations than advantages. Despite the Goppa codes promising a high security, the key sizes needed for a quantum-level of security are very big, resulting into a slow key establishment [37, p.47].

The NTRU protocol comes with many advantages. It is correct, which means that the IND-CCA2 KEM always establishes a key, therefore it never aborts because of a failure. The problem NTRU is based on, LWE, is very well studied and understood, giving NTRU an advantage in general knowledge on how the system works. Another advantage of NTRU cryptosystem is its simplicity. The Deterministic Public Key Encryption (DPKE) has only two parameters, n and q , and can be described entirely in terms of simple integer polynomial arithmetic. The transformation to an IND-CCA2 secure KEM is conceptually simple. A major disadvantage of NTRU

Tab. 3.2: Concluded advantages and limitations of the algorithms.

Algorithm	Advantages	Limitations
McEliece	high security of Goppa codes	big key sizes slow key establishment
NTRU	correct well studied simple	difficult to choose the right parameters
CRYSTALS-KYBER	fast well studied	slightly bigger key sizes
SABER	low-bandwidth relatively fast short key sizes	NTT not natively supported

is the difficulty of choosing the right parameters as is currently limited by a poor understanding of the non-asymptotic behavior of new algorithms for SVP. This is a limitation that is shared with all lattice based cryptosystems [44, p.36].

CRYSTALS-KYBER is based on LWE just like the NTRU protocol, making one of its advantages the fact that it is a well studied problem. Another major advantage is its fast key establishment, as measured on ASIC. A small disadvantage of this algorithm is its slightly bigger key sizes compared to other lattice based cryptosystems [33, p.28-30]. CRYSTALS-KYBER is a relatively new protocol and while the documentation when it comes to comparison with other protocols and expected security strength is very detailed, a general theoretical background seems to be slightly weak, making it harder to understand the general idea behind the protocol without an excessive research in various publications.

The SABER protocol requires only half of the bandwidth compared to NTRU or CRYSTALS-KYBER, thanks to its use of the MLWR problem. Although SABER was not as fast as CRYSTALS-KYBER during the key establishment process, it was only a matter of microseconds for the protocol to generate the keys on UltraScale+ FPGA. SABER also uses the shortest keys while still keeping the required level of security. The use of two-power moduli makes NTT⁴-like polynomial multiplication not natively supported. For this reason, SABER uses asymptotically slower polynomial multiplication algorithms such as ToomCook, Karatsuba, Schoolbook, or hybrids of them. A possible disadvantage of the protocol could also be the fact, that it is not capable of creating digital signatures, however for the purposes of this thesis which is focused solely on key-establishing mechanisms, this information does not make the protocol less valuable [35, p.22,23].

⁴Number Theoretic Transform Multiplication Algorithm (NTT). The NTT is a discrete Fourier transformation defined over a ring or a finite field and is used to multiply two integers without requiring arithmetic operations on complex numbers.

3.5.2 The Best One Is...

It is safe to say that McEliece with its big key sizes and slow key establishment is not the ideal candidate despite Goppa codes claiming to be very secure. As mentioned before, the only way McEliece would be ideal for use is if an application would continue using each public key for long enough to handle the costs of generating and distributing the key.

Protocols NTRU, CRYSTAL-KYBER and SABER are all based on lattices. All of the problems each cryptosystem is based on were compared and the SABER protocol seems like the best candidate, as it uses the MLWR problem which helps to cut the bandwidth to half compared to other lattice-based systems.

Unfortunately the information about the time needed to generate keys using NTRU was not available, which makes it harder to compare. The only hint about the speed of NTRU is in [44], mentioning that NTRU is unlikely to be the fastest candidate, along with stating that NTRU is also unlikely to be the most compact submission, and unlikely to be the most secure submission. With this information coming directly from the authors of the submission, in addition to the knowledge about RLWE, MLWE and MLWR obtained by now, it is safe to say that module-based problems are more efficient than ring-based problems. With this statement we can eliminate NTRU as the best candidate.

Cryptosystems CRYSTALS-KYBER and SABER are both very adequate, safe and fast protocols. CRYSTALS-KYBER appears to be faster, but more memory demanding than SABER. A notable advantage of SABER is the lower bandwidth required. Choosing from these two cryptosystems would be done only based on personal preferences, after establishing whether the algorithm needs to work fast with the cost of having to use more of the memory, or whether it is preferred to spare memory and having the algorithm work slightly slower.

For the further purposes of this thesis, we will deem CRYSTALS-KYBER as the most suitable candidate.

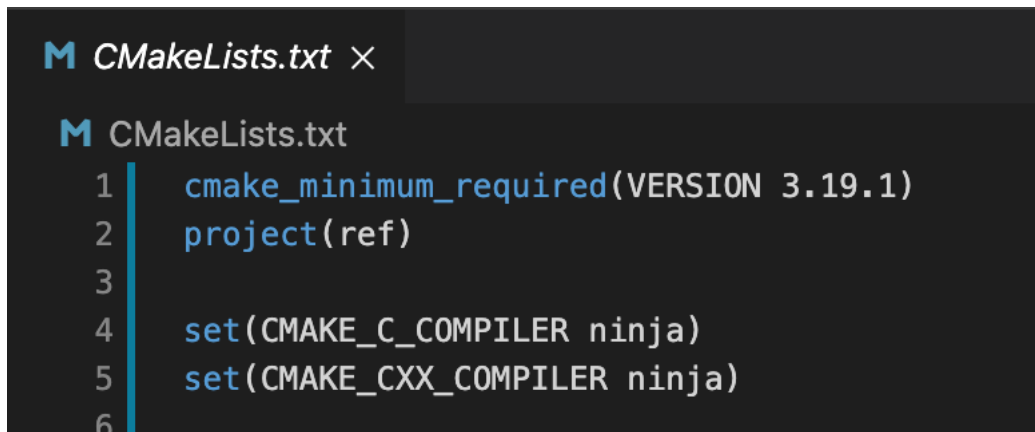
4 Implementation of CRYSTALS-KYBER

The implementation of the chosen protocol will be discussed on the following pages. The goal of this part of the thesis is for the protocol CRYSTALS-KYBER to be sufficiently set up and running.

4.1 Setting up the Environment

The source code of the protocol is available from the GitHub repository [46]. The first environment used to run the code was Visual Studio Code. After importing the code into the environment we followed the recommendation from the *readme* file to use the ninja build tool for a fast build performance. Ninja is a *cmake* based build system and due to the very little policy about how code is built, it is very fast [47]. The submitted code is indeed slightly chaotically structured, so it makes sense that the creators recommend using this tool.

After downloading and installing the *cmake* and *ninja* package, the first attempt for a build was carried out. Plenty of errors appeared, due to a wrong placement of the *CMakeLists.txt* file and a wrong configuration. The *CMakeLists.txt* file requires extra information that is not automatically included. In the Figure 4.1 we can see all the details that were necessary to be added.



```
M CMakeLists.txt ×  
  
M CMakeLists.txt  
1 cmake_minimum_required(VERSION 3.19.1)  
2 project(ref)  
3  
4 set(CMAKE_C_COMPILER ninja)  
5 set(CMAKE_CXX_COMPILER ninja)  
6
```

Fig. 4.1: Added information to CMakeLists.txt.

While the information and the location of the *CMakeLists.txt* file is now correct, after running the build we receive many new errors related to the original code in the above mentioned text file. All of the new errors are similar to this one *add_executable cannot create target "test_kyber_ref" because another target with the same name already exists. The existing target is an executable created in source directory*, with the difference in the specific target. After checking however, the

mentioned executable nor any file named the same exists in the directory. Many sources offered a different approach of fixing a similar problem, none of them were unfortunately successful. Visual Studio Code is also unable to read some of the classes, due to the code not being written in one uniform language. In order to fix the problem, we will try to run the code in a different IDE.

For the next attempt on running the source code CLion was used. After importing the code to CLion a cmake build was automatically generated (Figure 4.2), without having to manually download and install any extra tools. CLion IDE also recognizes all of the files of the code. The whole code is now buildable and runnable.

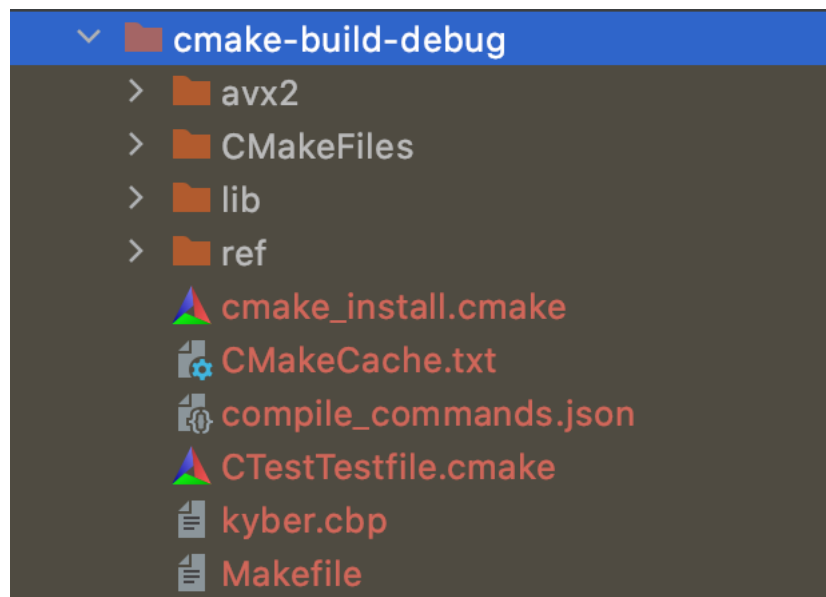


Fig. 4.2: Automatically generated cmake build in CLion.

4.2 Exploring the Capabilities of the Code

As mentioned before in Chapter 3.3, CRYSTALS-KYBER is a protocol using a key encapsulation mechanism to establish a shared key k .

The existing source code contains variety of classes executing vector calculations, NTT algorithm, generation of a private and a public key, key encapsulation, key decapsulation and also the shared key establishment.

The code has multiple parameters, most of them are defined in *params.h*. The part of the code of the header file is shown below (Listing 4.1). *KYBER_SYMBYTES* defines the size for hashes and seeds, *KYBER_SSBYTES* defines the size of the shared key. Some of the parameters contain comments, most of them however do not. A documentation to the source code does not exist, therefore for some of the

parameters it is impossible to surely deduce what they are. We suppose, that *KYBER_POLYBYTES* defines the size of the polynomial and *KYBER_POLYVECBYTES* defines a size of the vectors for lattices. The number of bytes of the public key is a sum of the size of a vector in a lattice and the size of the seed. The size of the secret key is equal to the size of the vector in a lattice.

```

1 #define KYBER_SYMBYTES 32    /* size in bytes of hashes, and seeds */
2 #define KYBER_SSBYTES 32    /* size in bytes of shared key */
3
4 #define KYBER_POLYBYTES 384
5 #define KYBER_POLYVECBYTES (KYBER_K * KYBER_POLYBYTES)
6
7 #define KYBER_INDCPA_PUBLICKEYBYTES (KYBER_POLYVECBYTES +
   KYBER_SYMBYTES)
8 #define KYBER_INDCPA_SECRETKEYBYTES (KYBER_POLYVECBYTES)

```

Listing 4.1: Code: Parameters in *params.h*.

In different parts of the code, the value of the seed is hard coded. In the *test_speed.c* (closely discussed later, the output of the function can be seen on figure 4.5), the value of the seed is set to 0 on all 32 positions. The seed used for actual generation of the keys shown in Figure 4.6 is defined on line 3 of the Listing 4.2. It is also hard to understand, why the authors did not use the value of *KYBER_SYMBYTES* (4.1) to define the size of the array, as it would simplify the scalability of the protocol and add a purpose to defining this parameter in *params.h*.

```

1 uint8_t seed[KYBER_SYMBYTES] = {0};
2
3 static uint32_t seed[32] =
   {3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3,8,4,6,2,6,4,3,3,8,3,2,7,9,5};

```

Listing 4.2: Code: Defined values of a seed.

The source code contains different test classes. Only the test classes contain main functions. The *test_kex.c* tests the functionality of the key exchange and as an output it returns the number of transferred bytes during the key establishment (as shown in the Figure 4.3).

```

KEX_UAKE_SENDABYTES: 2272
KEX_UAKE_SENDBBYTES: 1088
KEX_AKE_SENDABYTES: 2272
KEX_AKE_SENDBBYTES: 2176

```

Fig. 4.3: Output: Bytes sent during the key establishment.

A *kyber_test.c* tests the functionality of the entire protocol. When calling this function, at first the process of generating the keys is executed (described in the Chapter 3.3). Alice generates her pair of keys (public and secret key). Bob then derives a secret key to get his shared key and creates a response, that is sent to Alice. Alice then proceeds to use Bobs response to generate her shared key. The code then proceeds to compare the values of the shared keys. If the values are equal the output of the class *kyber_test.c* is numbers of bytes generated for the keys, as seen in the Figure 4.4.

```
CRYPTO_SECRETKEYBYTES: 2400
CRYPTO_PUBLICKEYBYTES: 1184
CRYPTO_CIPHERTEXTBYTES: 1088
```

Fig. 4.4: Output: Generated key bytes.

From the *test_speed.c* the expected output would logically be the time it takes to run specific parts of the code on the used device or an information about how many *bits/second* are processed. The output however returns values of *cycles/ticks*. A tick is an arbitrary unit for measuring internal system time, however some authors also use tick as a synonym for processor clock cycle. The official documentation does not specify what exactly a tick is in this case, as it is also not readable from the code. When it comes to cycles, specifically cycles related to key establishment, the system does 10 000 rounds of the key establishment process by default. This number can be easily adjusted based on the users preferences in *cpucycles.c*. As an example of the output for a speed of establishing the shared keys can be seen in the figure 4.5.

```
kex_ake_sharedB:
median: 7114890 cycles/ticks
average: 7410592 cycles/ticks

kex_ake_sharedA:
median: 5025276 cycles/ticks
average: 5260100 cycles/ticks
```

Fig. 4.5: Output: Speed of establishing the shared keys.

The shortcut *AKE* stands for a mutually authenticated key exchange. The source code also contains a code for establishing a unilaterally authenticated key exchange.

The last test file is *test_vectors.c*. This file executes the whole process of key generation, encapsulation and a shared key establishment. More about the key establishment in the code implementation is in the separate chapter below (Chapter 4.3).

4.3 Key Establishment

The key establishment uses the process of key generation, key encapsulation and key decapsulation in order to create a mutually agreed shared key. The process of a shared key establishment is described in Chapter 3.3.

The key establishment in the code follows these simplified steps:

- Alice generates a matrix A using the seed and sends the value of a seed to Bob.

```
1 static uint32_t seed[32] =  
    {3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3,8,4,6,2,6,4,3,3,8,3,2,7,  
2 9,5};  
3
```

Listing 4.3: Seed parameter for generating the matrix.

- Alice generates a key pair of predefined size, the code for generation is shown in the Listing 4.9. Along with this generation a transpose of the matrix A is created.
- Bob also generates his key pair the same way as Alice.
- Alice sends her public key to Bob and Bob sends his public key to Alice.
- Alice encapsulates Bob's public key with her secret and sends this value to Bob.

```
1 crypto_kem_enc(send+CRYPTO_PUBLICKEYBYTES, tk, pkb);  
2
```

Listing 4.4: Called encapsulation function on Alice's side.

- Bob encapsulates Alice's public key with the encapsulated information he received from her and sends it back.

```
1 crypto_kem_enc(send+CRYPTO_CIPHERTEXTBYTES, buf+CRYPTO_BYTES,  
    pka);  
2
```

Listing 4.5: Called encapsulation function on Bob's side.

- Bob decapsulates the received information using his secret key and creates a shared key.

```

1  crypto_kem_dec ( buf+2*CRYPTO_BYTES,  recv+CRYPTO_PUBLICKEYBYTES,
2  skb );

```

Listing 4.6: Called decapsulation function on Bob’s side.

- Alice decapsulates the recieved information with her secret key and creates a shared key.

```

1  crypto_kem_dec ( buf+CRYPTO_BYTES,  recv+CRYPTO_CIPHERTEXTBYTES,
2  ska );

```

Listing 4.7: Called decapsulation function on Alice’s side.

- The shared key is verified using a shake256.

```

1  shake256 (k,  KEX_SSBYTES,  buf,  3*CRYPTO_BYTES);
2

```

Listing 4.8: Called shake function for verification.

All of the called functions mentioned above are included in the class *indcpa.c*. Below a part of this c-file that includes the function for key generation is shown. As the rest of the code is longer, complicated, points to a variety of different places in the code and therefore would not offer valuable information, it was not included in this thesis. The code is however available at the official GitHub Repository [46].

```

1  void indcpa_keypair (uint8_t pk[KYBER_INDCPA_PUBLICKEYBYTES] ,
2                      uint8_t sk[KYBER_INDCPA_SECRETKEYBYTES])
3  {
4      unsigned int i;
5      uint8_t buf[2*KYBER_SYMBYTES];
6      const uint8_t *publicseed = buf;
7      const uint8_t *noiseseed = buf + KYBER_SYMBYTES;
8      polyvec a[KYBER_K], e, pkpv, skpv;
9
10     randombytes (buf, KYBER_SYMBYTES);
11     hash_g (buf, buf, KYBER_SYMBYTES);
12
13     gen_a (a, publicseed);
14
15     #ifndef KYBER_90S
16     #define NOISE_NBLOCKS ((KYBER_ETA1*KYBER_N/4)/AES256CTR_BLOCKBYTES) /*
17         Assumes divisibility */
18     uint64_t nonce = 0;
19     ALIGNED_UINT8(NOISE_NBLOCKS*AES256CTR_BLOCKBYTES+32) coins; // +32
20     bytes as required by poly_cbd_eta1
21     aes256ctr_ctx state;
22     aes256ctr_init (&state, noiseseed, nonce++);

```



```

21  for (i=0;i<KYBER_K;i++) {
22      aes256ctr_squeezeblocks (coins.coeffs , NOISE_NBLOCKS, &state);
23      state.n = __mm_loadl_epi64((__m128i *)&nonce);
24      nonce += 1;
25      poly_cbd_eta1(&skpv.vec[i] , coins.vec);
26  }
27  for (i=0;i<KYBER_K;i++) {
28      aes256ctr_squeezeblocks (coins.coeffs , NOISE_NBLOCKS, &state);
29      state.n = __mm_loadl_epi64((__m128i *)&nonce);
30      nonce += 1;
31      poly_cbd_eta1(&e.vec[i] , coins.vec);}

```

Listing 4.9: Key generation.

The class *test_vectors.c* does by default 10 000 rounds of key generation, encapsulation and shared key establishment. The result of one of the cycles is shown in Figure 4.6. The public (the size of the public key in this case is 1 568 bytes) and secret keys (the size of the secret key in this case is 1 535 bytes) are cut off due to a better readability of the shared secrets. The Shared Secret A is identical to the Shared Secret B, it is therefore confirmed, that the code works correctly.

```

Public Key: 7b5242398086b7c98b64572c6e16c64dd5c1b06350ff48077af17ab5d003e877b0ad229
Secret Key: a6a3b0a119c740e5bb20077d436172f966cb4885807f7ac43dfbaa82f75d9b04a10f9b8
c571e3a396926071b6a6363f018e0d2ffed38d9057b4660226742eb762ebef86
Ciphertext: 8a55c37b887b89e4fb1b65d1a7a64e4375cadf4b434018247c58e4d394928bdd1bd442f
Shared Secret B: 42507624c5a1bdd2beb303451d79cc2184f6eaec1b10eea4b760004b0c692dd4
Shared Secret A: 42507624c5a1bdd2beb303451d79cc2184f6eaec1b10eea4b760004b0c692dd4

```

Fig. 4.6: Output: Generated keys.

As all of the input values (for example the seed) are predefined, the output values of the source code are going to be the same every time the code is ran. This solution would not work very well for a practical use and would dramatically lower the actual security of the protocol. However, for the testing purposes the static input values reduce the time needed to build and execute the code.

4.4 Time Needed to Establish the Keys

As the *test_speed.c* does not return the information that we would like to know about the speed of the key establishment, the next step would be adjusting the code that returns the information in desired form - *bits/second*.

The file *test_vectors.c* runs the entire process of key establishment, therefore the additional function that calculates the time needed for establishment was implemented in this class. The Listing 4.10 shows the added code, that uses the clock

function which is included in C language. This function then measures the time needed to execute the key establishment for *NTSEST* cycles, which was 10000 in this case.

The time needed to execute one round of key establishment is then calculated as an average. The original source code has the number of bytes used for the establishment predefined and stored in *KEX_AKE_SENDABYTES*. This value is then converted into bits.

The code then returns the information about the time needed to execute *NTSEST* number of cycles, bits sent per one cycle, an average time needed for one cycle, how many bits per second are generated and how long it takes to generate a single bit.

The time measured by the added code is a CPU time. CPU time does not take into consideration the time when the application is waiting for inputs or outputs. In this case, no wait was involved, therefore the time measured by the code (CPU time) and the real time it takes to generate keys are very similar.

```
1 void speed_keyestab () {
2     unsigned int i;
3     unsigned char pk[CRYPTO_PUBLICKEYBYTES];
4     unsigned char sk[CRYPTO_SECRETKEYBYTES];
5     unsigned char ct[CRYPTO_CIPHertextBYTES];
6     unsigned char key_a[CRYPTO_BYTES];
7     unsigned char key_b[CRYPTO_BYTES];
8
9     clock_t time;
10    time = clock();
11
12    for (i=0;i<NTESTS;i++) {
13        // Key-pair generation
14        crypto_kem_keypair(pk, sk);
15
16        // Encapsulation
17        crypto_kem_enc(ct, key_b, pk);
18
19        // Decapsulation
20        crypto_kem_dec(key_a, ct, sk);
21    }
22
23    time = clock() - time;
24    double time_taken = ((double)time) / CLOCKS_PER_SEC;
25    double time_cycle = time_taken / NTESTS;
26    int bits = KEX_AKE_SENDABYTES * 8;
27    int bits_per_sec = bits / time_cycle;
28    double time_per_bit = (time_cycle / bits) * 100000;
29
30    printf("\n*** RESULTS ***\n\n");
```

```

31     printf("%d cycles finished after: %f seconds\n", NTESTS, time_taken
);
32     printf("Bits sent per cycle: %d\n", bits);
33     printf("%f seconds/cycle\n", time_cycle);
34     printf("%d bits/second\n", bits_per_sec);
35     printf("%.10f microseconds needed per bit\n", time_per_bit);
36 }

```

Listing 4.10: Added lines of code: Key establishment time.

In order to make the time measuring more accurate, the printing of the generated keys was paused. The keys were generated in the background and the only output are the results of the newly created speed test.

The time measuring was carried out on a MacBook Pro with 2,7 GHz Dual-Core Intel Core i5 and 8 GB 1 865 MHz DDR3 SDRAM. The results of the measuring are shown in Figure 4.7.

The time needed to generate a single bit is very low, at only 0,0 113 463 030 μ s. The average time needed to establish the keys on the above mentioned device was 0,002 847 seconds, being 2 847 μ s. Compared to a claimed time needed to establish the key on an ASIC being at 1 500 μ s (mentioned in the Table 3.1), the value measured on a personal computer is higher. This difference is understandable, as the ASIC is a chip used for one specific purpose, whereas a personal computer runs multiple operations at the same time.

```

*** RESULTS ***

10000 cycles finished after: 28.465605 seconds
Bits sent per cycle: 25088
0.002847 seconds/cycle
8813443 bits/second
0.0113463030 microseconds needed per bit

```

Fig. 4.7: Output: Key establishment time (Test1).

The computer used for measuring the time needed to establish the keys above is relatively old, to see how the performance would be affected by a more sufficient device, a newer version of a MacBook Pro was used.

The second test was carried out on a MacBook Pro with 2,4 GHz 8-Core Intel Core i9 and 32 GB 2 400 MHz DDR4 SDRAM. The results of the measuring can be seen in Figure 4.8.

```
*~*~*~*~* RESULTS *~*~*~*~*

10000 cycles finished after: 18.082824 seconds
Bits sent per cycle: 25088
0.001808 seconds/cycle
13873939 bits/second
0.0072077583 microseconds needed per bit
```

Fig. 4.8: Output: Key establishment time (Test2).

The time needed to generate a single bit is very low, at only $0,0072077583 \mu\text{s}$. The average time needed to establish the keys on the newer, more sufficient device was $0,001808$ seconds, being $1808 \mu\text{s}$.

The newer MacBook Pro was capable of establishing the keys in the time that is comparable to the time needed to establish the keys on an ASIC ($1500 \mu\text{s}$). Considering that the personal computer runs multiple operations in the background at the same time, it is a remarkable speed.

4.5 Suggested Adjustments

The standing documentation regarding the source code of CRYSTALS-KYBER protocol [33] does not offer a detailed description to understand how the protocol works, nor does it offer a user manual. The documentation also misses some key explanations related to the theory of the protocol. It is understandable, that the submission makes sense to professionals even without the extra detailed information, however, if the protocol becomes standardized, the interest will surely also come from the direction of ordinary people.

The next step of this thesis therefore is a creation of a simplified CRYSTALS-KYBER application, a documentation of the adjusted application and an easily understandable user manual.

5 Adapted Implementation of CRYSTALS-KYBER

This chapter covers the process of the CRYSTALS-KYBER simplification and optimization for a possible every-day use.

5.1 Changes and Specifications

The original CRYSTALS-KYBER code from the NIST submission only mimics the key-agreement process in one single application, it does not allow for two parties to agree on the key if they were physically in different places. The new implementation should allow a key agreement for two parties through a local IP connection.

In this case, Alice and Bob communicate over a TCP connection. The process of sending the information and establishing the shared key is shown in the picture below (5.1). Calculations are described in more detail in Chapter 3.3.

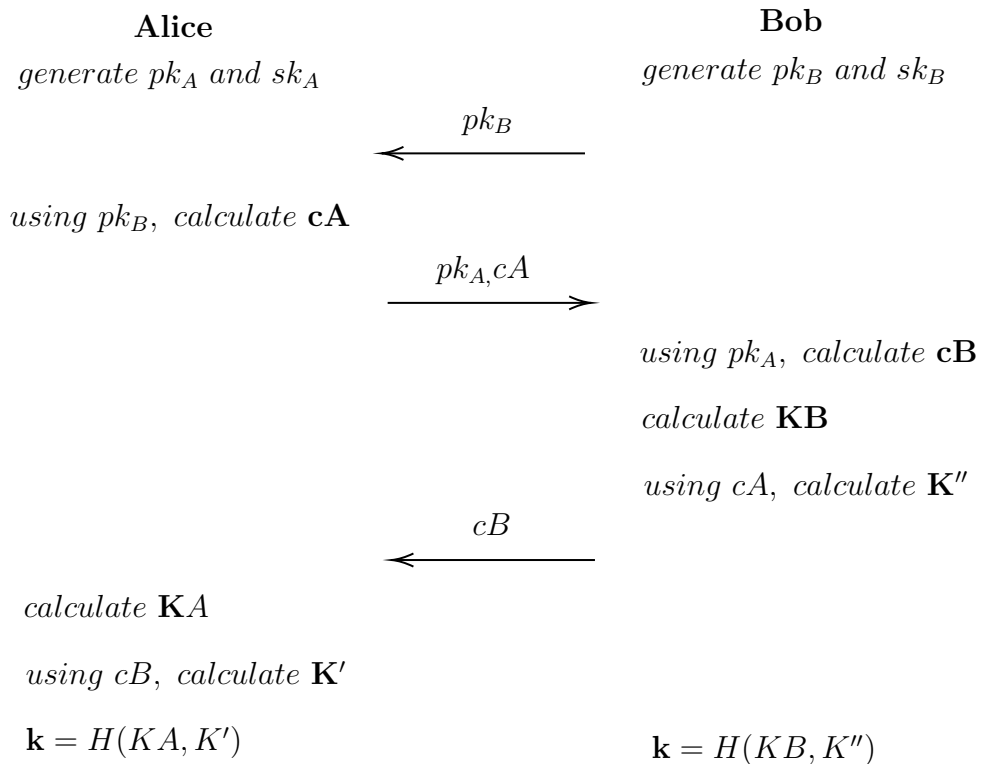


Fig. 5.1: Key establishment process.

For the above mentioned to work, the application needed to be separated - Alice and Bob both needed to have an individual code, that would allow them to do the calculations after receiving the necessary information from the other party. The

flowchart of the new code for Alice and Bob can be found on the following pages (flowchart for Alice is in the Figure 5.2 and for Bob in the Figure 5.3). In the context of the TCP communication, Alice poses as a Server and Bob represents the Client. Alice creates a socket and waits for Bob to connect. After accepting the Client, the key establishment is executed.

The new implementation works with the necessary libraries from the original NIST submission package [48] and consists of multiple parts. Those are specifically:

- Shell scrips for Server and Client that take the parameters and execute the key establishment based on them.
- A code for Alice
 - TCP connection code for Server
 - Code for CRYSTALS-KYBER
- A code for Bob
 - TCP connection code for Client
 - Code for CRYSTALS-KYBER

The codes for Alice and Bob were used to create executables that are called by the mentioned shell scripts. Executables that are a part of this project were created for macOS BigSur version 11.1 and higher. The other version of the executables attached is runnable on Ubuntu 20.04.2. Every executable that is created using cmake is adjusted to the operational system that it was created on. If a user wishes to create new executables for a different operational system, attachment C provides simple instructions on how to do so.

The newly created TCP implementation of CRYSTALS-KYBER was also made publicly available on GitHub [49].

Fig. 5.2: Flowchart for Alice's part of the key exchange.

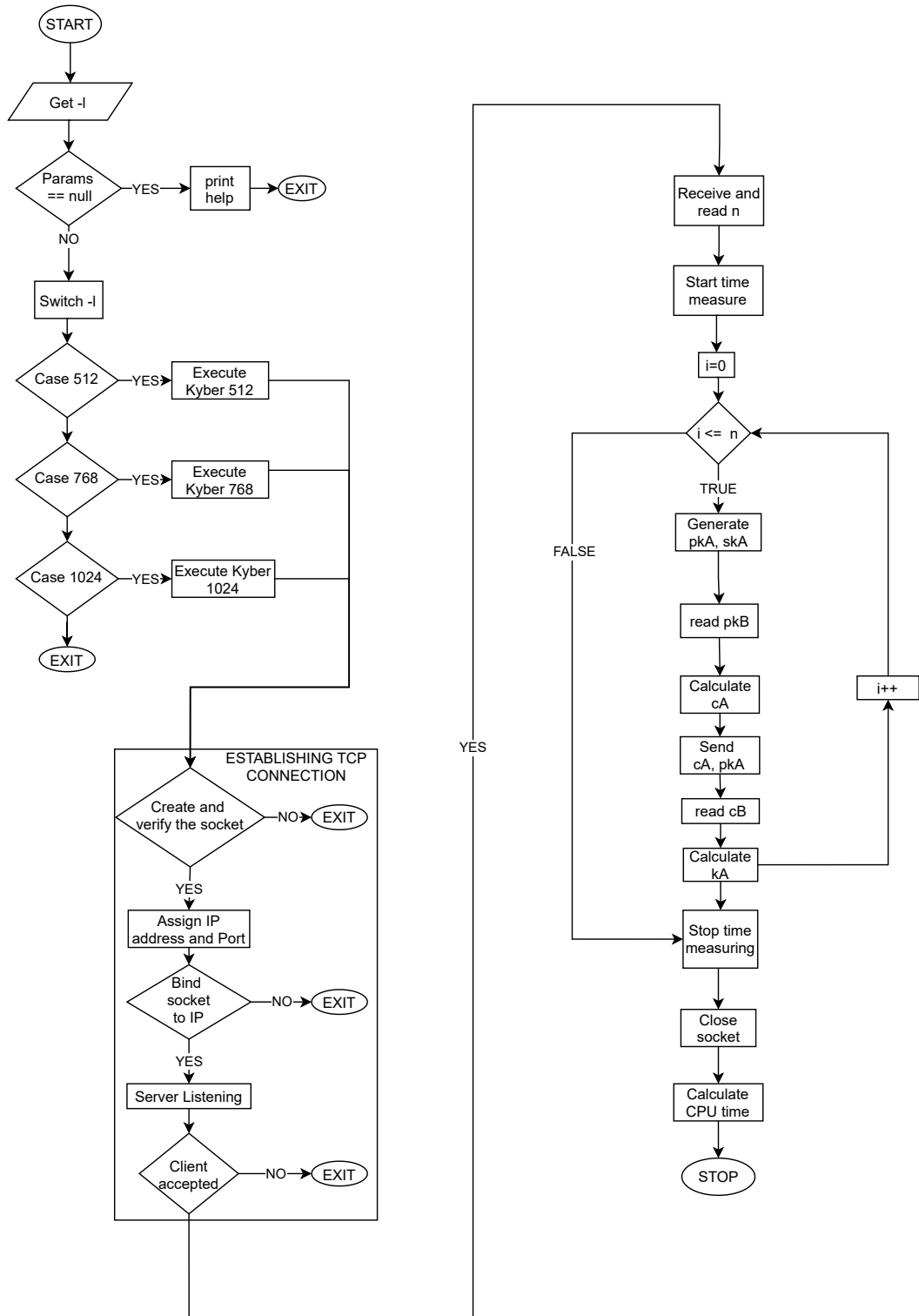
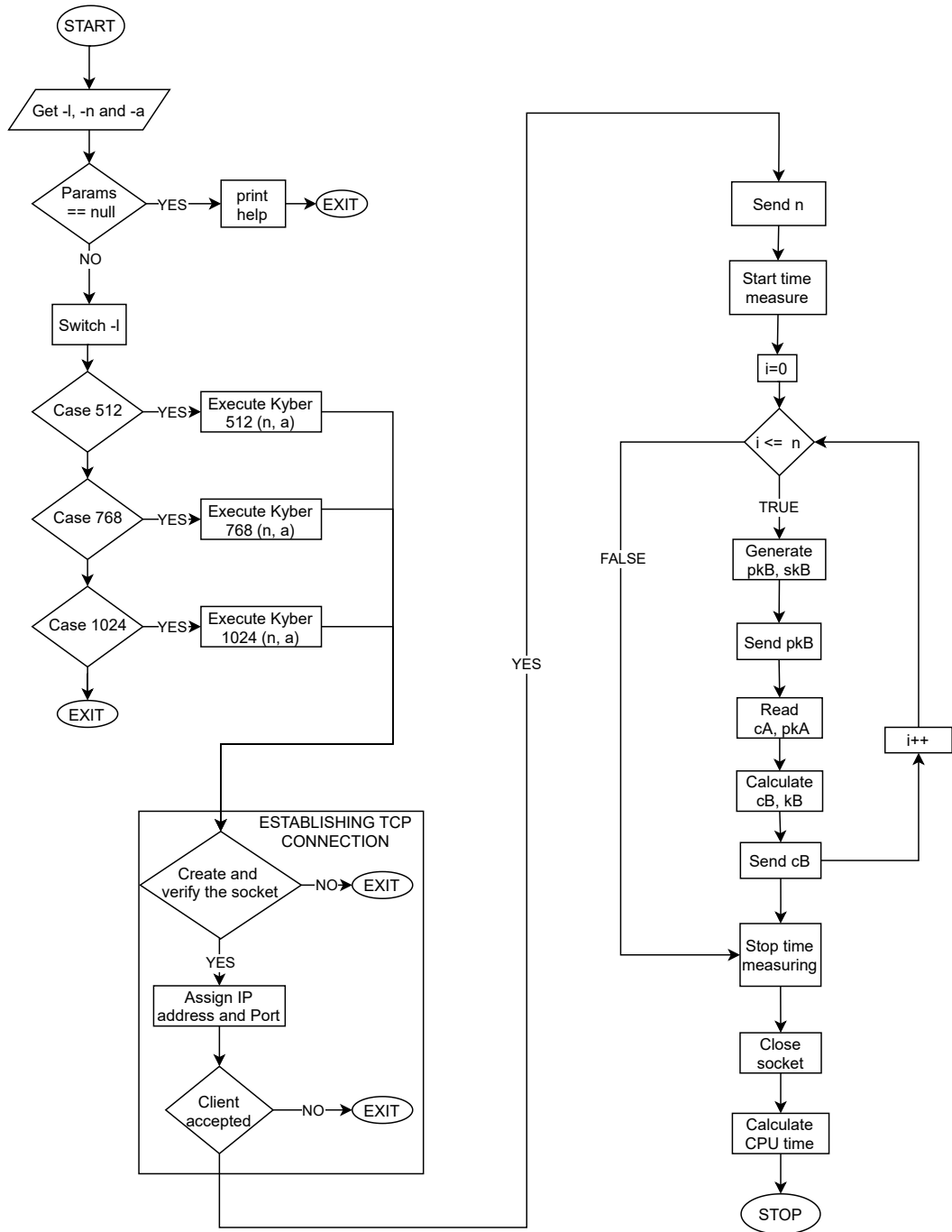


Fig. 5.3: Flowchart for Bob's part of the key exchange.



5.2 Use of the New Implementation

If a user wishes to use the newly created implementation, the process is simple. The first step is to open a terminal window for a Server or a Client and direct themselves to a directory that contains the executable files. In the example scenario, both Alice (Server) and Bob (Client) are executed on one computer.

After typing in the request for running the shell script, the user gets a brief explanation of what the application does and how to work with it (shown in Figures 5.4 and 5.5).

```
Tatianas-MacBook-Pro:Executables tnovosadova$ ./KyberServer
usage: ./KyberServer options

This is an application for establishing a shared key based on Crystals-Kyber protocol. It is
necessary to RUN THE SERVER FIRST. Parameter for security level needs to match on both sides.

Example of a command for Server: ./KyberServer -l 1024

OPTIONS:
  -h      Show this message
  -l      Security level, can be 512, 768 or 1024
```

Fig. 5.4: Terminal window server - input the parameters.

```
Tatianas-MacBook-Pro:Executables tnovosadova$ ./KyberClient
usage: ./KyberClient options

This is an application for establishing a shared key based on Crystals-Kyber protocol. It
is necessary to RUN THE SERVER FIRST. Parameter for security level needs to match on both
sides.

Example of a command for Client: ./KyberClient -l 1024 -n 100 -a 127.0.0.1

OPTIONS:
  -h      Show this message
  -l      Security level, can be 512, 768 or 1024
  -n      Number of Cycles to be executed, has to be > 0
  -a      IPv4 of Alice (Server)
```

Fig. 5.5: Terminal window client - input the parameters.

As mentioned in the Chapter 3.3, CRYSTALS-KYBER offers three levels of security, where Kyber-512 aims at security roughly equivalent to AES-128, Kyber-768 aims at security roughly equivalent to AES-192, and Kyber-1024 aims at security roughly equivalent to AES-256 [33]. The application therefore also allows a choice for the desired level of security by simply setting the parameter `-l`. This parameter needs to match on Server's and Client's side.

All the other parameters (number of cycles `-n` and the IPv4 address of the server `-a`) are required to be entered only on Client's side. The number of cycles can be

any positive integer. If both Client and Server are ran on one device, the address of the Server will be a local-host address.

It is necessary to run the application for the Server first. The Server then waits for the Client to connect (Figure 5.6).

```
Tatianas-MacBook-Pro:Executables tnovosadova$ ./KyberServer -l 1024
You choose Kyber1024
Socket successfully created..
Socket successfully bound..
Server listening..
```

Fig. 5.6: Terminal window server - server listening.

In the second terminal window the user enters the parameters for the Client. In this case, the number of cycles is set to 1, therefore only one shared key is generated. After successfully connecting to the Server, both sides start the Key Generation process as described in Figure 5.1.

```
Tatianas-MacBook-Pro:Executables tnovosadova$ ./KyberClient -l 1024 -n 1 -a 127.0.0.1
You choose Kyber1024
Socket successfully created..
Connected to server (Alice)...

Generating keypair...
```

Fig. 5.7: Terminal window client - start of the key generation.

The Key Generation then runs automatically and results with outputting a successfully calculated shared key (5.8).

```
Shared Key Alice for cycle 1:
cb7879f345650c6f5d1d3860305d210d069b77375f4e3bfd79e6efcb1d432b75

Shared Key Bob for cycle 1:
cb7879f345650c6f5d1d3860305d210d069b77375f4e3bfd79e6efcb1d432b75
```

Fig. 5.8: Successfully calculated shared key.

The entire key generation is very fast, however if the user wishes to take a look at the information calculated and sent during the process, everything is listed in the terminal window above the final shared key.

The application also calculates the time needed to establish the keys. This part is covered in more detail in the following chapter (Chapter 5.3).

5.3 Time Needed to Establish the Keys

The option to calculate the time needed to establish the keys was added into the original NIST submission (4.4). The function was also implemented in the new application. This time however, the time is calculated separately for Alice and Bob. As shown in the flowcharts 5.2 and 5.3, the measuring of the time only starts after the Client is successfully accepted by the Server.

As in the previous implementation (Chapter 4.4), the application calculates CPU time instead of a real time. The decision to do so was based on the amount of the data that needs to get printed out. The CPU time does not include the time taken to print out an output. Therefore the time shown in the end only shows the time needed to calculate the shared-keys in the given number of cycles.

In the chapter 4.4, the time was measured for the implementation of Kyber-1024 and 10 000 cycles. The time needed for 10 000 cycles was also measured for the new implementation, for all of the available security levels. The time results are automatically calculated after executing the chosen number of cycles and displayed as an output in the terminal, as shown in the Figure 5.9.

```
Number of cycles: 10000.  
CPU time measured Alice: 53.55971 seconds.  
Average CPU time per cycle Alice: 0.00536 seconds.  
Number of cycles: 10000.  
Total CPU time measured Bob: 48.29086 seconds.  
Average CPU time per cycle Bob: 0.00483 seconds.
```

Fig. 5.9: Time needed to establish the keys for Alice and Bob separately.

The measured time is almost two-times higher than within the original application, which makes sense considering all the extra steps that are done in the new implementation (sending the values to each other through the TCP communication).

The time was measured on two personal computers with different properties. Results of the tests and the devices are more closely discussed below.

Values shown in the Table 5.1 were measured on a MacBook Pro with 2,7 GHz Dual-Core Intel Core i5 and 8 GB 1 865 MHz DDR3 SDRAM.

The second test was carried out on a MacBook Pro with 2,4 GHz 8-Core Intel Core i9 and 32 GB 2 400 MHz DDR4 SDRAM. Table 5.2 shows the results of this test.

Alice always takes longer than Bob to generate her shared key. This is caused by the fact, that Bob calculates his shared key before sending the last parameter that Alice needs to calculate hers. This part cannot be optimized, because Bob calculates the last parameter in the same step as he creates his shared key.

Tab. 5.1: The time needed to establish the keys based on the level of security (1).

Level of Security	Alice (time/10 000 cycles) [s]	Bob (time/10 000 cycles) [s]
Kyber-512	26.25495	24.53068
Kyber-768	40.06725	35.96198
Kyber-1024	53.55971	48.29086

Tab. 5.2: The time needed to establish the keys based on the level of security (2).

Level of Security	Alice (time/10 000 cycles) [s]	Bob (time/10 000 cycles) [s]
Kyber-512	13.97371	12.35899
Kyber-768	20.98202	18.70328
Kyber-1024	28.51103	25.66362

The newer MacBook Pro was capable of establishing the keys in a relatively similar time as the original application.

Even though the length of the shared keys is always the same (thanks to the hash function at the end of all the operations), the time grows with the higher level of security. The reason for this is the changing length of the values needed to establish the keys.

Conclusion

This master's thesis had multiple goals. One of the objectives was to introduce the topics of quantum computers, post-quantum cryptography and the principles different types of the post-quantum cryptography are based on. The main goal was to study, analyze and compare the submission of the third round of the NIST competition (introduced in Chapter 3), with an intention to find the best candidate. The last objective of the thesis was to create an optimized application for key establishment based on the chosen algorithm, along with creating an installation guide and a user manual.

The theoretical part of the thesis introduces the post-quantum cryptography based on hashes, codes, lattices, supersingular elliptic curves and a multivariate cryptography. As the NIST finalists were cryptosystems based on codes and lattices, these types of post-quantum are described in greater detail than the others. The cryptosystems McEliece, NTRU, CRYSTALS-KYBER and SABER are analyzed in depth. These cryptosystems use different unsolvable problems for reaching a desired quantum security. The problems these cryptosystems are based on were introduced and described both in formal and informal, simplified way. The thesis explained the syndrome decoding of linear error-correcting codes in Chapter 3.1, a learning with errors problem (Chapter 3.2) and its modification using modules (Chapter 3.3). A learning with a rounding problem was also discussed in Chapter 3.4.

The available documentation related to the finalists of the NIST competition was used and compared them in Chapter 3.5. Based on this part, the most suitable candidate for the key establishment protocol was decided to be the CRYSTALS-KYBER protocol. Protocol CRYSTALS-KYBER was defined as the fastest out of all the finalists, with the reasonable key sizes and offering the desired level of security.

The existing implementation of the CRYSTALS-KYBER was set up, analyzed and tested. While the recommended build tool (ninja) did not prove to be a successful tool for building the application, CLion ran the code without issues. The protocol is capable of establishing keys in a reasonable time. A function for measuring the time needed to establish keys was successfully implemented to the original source code and the speed of establishment was compared on two different devices. While the time needed to establish the keys on a personal computer is higher than on an ASIC, the results were still adequate. The original application from the NIST submission however only mimics the key-agreement process in one single application, it does not allow for two parties to agree on the key if they were physically in different places.

In the Chapter 5.3 an adapted implementation of CRYSTALS-KYBER was pro-

posed. This implementation allows for a key establishment over a TCP connection, allowing a user to choose the desired level of security, as well as the number of cycles to be ran. The adapted implementation successfully generates the shared keys. The time for establishing the keys was measured for the new application in a similar manner as in the original code. The new implementation takes longer to establish the keys, however this is understandable, due to the extra steps that are connected to the TCP communication.

The adapted application was described and documented in Chapter 5.1 along with visual flowcharts for an easier understanding of the functionality. The Chapter 5.2 describes how a user can execute the application in a few simple steps. Additionally, an installation guide (A) and a user manual (B) were added to the appendices.

The instructions for this thesis also called for a usable graphical user interface (GUI). This part was after consideration left out. The application aims to be simple and minimalist, focusing on the functionality rather than appearance. Even without a GUI, the created application is easy to use and understand, as well as the output data is clear to read.

The goals of the master's thesis were reached. Theoretical background of the post-quantum cryptography was introduced, the NIST finalists were analyzed and compared. The most suitable candidate was chosen and a software implementation that was created based on the given algorithm successfully establishes the shared key k .

Bibliography

- [1] NIST *Post-Quantum Cryptography Standardization NIST Information Technology Laboratory COMPUTER SECURITY RESOURCE CENTER* [online], 2020 [cit. 2020-10-15]. Available from: <<https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions>>
- [2] *Quantum Superposition*, 2020. Joint Quantum Institute [online]. [cit. 2020-11-15]. Available from: <<https://jqj.umd.edu/glossary/quantum-superposition>>
- [3] FEYNMAN, Richard P. *Simulating Physics with Computers* [online]. Pasadena, California, 1981 [cit. 2020-10-18]. Available from: <<https://web.archive.org/web/20190108115138/https://people.eecs.berkeley.edu/~christos/classics/Feynman.pdf>> Department of Physics, California Institute of Technology.
- [4] *The Holevo bound*. Quantiki [online]. 2018 [cit. 2020-10-18]. Available from: <<https://quantiki.org/wiki/holevo-bound>>
- [5] GROVER, Lov K. *A fast quantum mechanical algorithm for database search: Grover's algorithm* [online]. Murray Hill NJ 07974 [cit. 2020-10-18]. Available from: <<https://arxiv.org/pdf/quant-ph/9605043v3.pdf>> Bell Labs.
- [6] SHOR, P.W. *Algorithms for quantum computation: discrete logarithms and factoring*[online]. Santa Fe, NM, USA, USA, 1994 [cit. 2020-10-18]. Available from: <<https://ieeexplore.ieee.org/document/365700>>
- [7] CHUANG, Isaac L., Neil GERSHENFELD a Mark KUBINEC. *Experimental Implementation of Fast Quantum Searching* [online]. California|Massachusetts, 1998 [cit. 2020-10-18]. Available from: <https://pdfs.semanticscholar.org/6c05/5053f4f1605fdc0bd474c7a350dcd01f627d.pdf?_ga=2.71539539.1715281349.1603457478-145387920.1603457478> Physical Review Letters. IBM Almaden Research Center, Physics and Media Group, MIT Media Lab, Cambridge, College of Chemistry: University of California
- [8] JOHNSTON, Hamish. *Quantum-computing firm opens the box. Physics World*[online]. [cit. 2020-10-18]. Available from: <<https://web.archive.org/web/20110515083848/http://physicsworld.com/cws/article/news/45960>>

- [9] CHATTERJEE, Dibyendu and ROY, Arijit. *A transmon-based quantum half-adder scheme*. *Progress of Theoretical and Experimental Physics* [online]. September 2015. Vol. 2015, no. 9, p. 093A02. [cit. 2020-10-18]. DOI 10.1093/ptep/ptv122. Available from: <<https://paperity.org/p/73955611/a-transmon-based-quantum-half-adder-scheme>>
- [10] PORTER, Jon. *Google may have just ushered in an era of ‘quantum supremacy*. *The Verge* [online]. 23 September 2019. [cit. 2020-10-18]. Available from: <<https://www.theverge.com/2019/9/23/20879485/google-quantum-supremacy-qubits-nasa>>
- [11] LERNER Louise, *UChicago scientists discover way to make quantum states last 10,000 times longer* | Argonne National Laboratory, 2020. Anl.gov [online]. [cit. 2020-11-16]. Available from: <<https://www.anl.gov/article/uchicago-scientists-discover-way-to-make-quantum-states-last-10000-times-longer>>
- [12] YANG, C. H., LEON, R. C. C., HWANG, J. C. C., SARAIVA, A., TANTTU, T., HUANG, W., CAMIRAND LEMYRE, J., CHAN, K. W., TAN, K. Y., HUDSON, F. E., ITOH, K. M., MORELLO, A., PIORO-LADRIÈRE, M., LAUCHT, A. and DZURAK, A. S., 2020. *Operation of a silicon quantum processor unit cell above one kelvin* [online]. April 2020 [cit. 2020-11-16]. DOI 10.1038/s41586-020-2171-6. Available from: <<https://arxiv.org/pdf/1902.09126.pdf>>
- [13] CHEN, Lily, JORDAN, Stephen, LIU, Yi-Kai, MOODY, Dustin, PERALTA, Rene, PERLNER, Ray and SMITH-TONE, Daniel, 2016. *Report on Post-Quantum Cryptography* [online]. April 2016. [cit. 2020-10-18]. DOI 10.6028/nist.ir.8105. Available from: <<https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf>>
- [14] *Conferences*, 2020. Pqcrypto.org [online]. [cit. 2020-10-20]. Available from: <<https://pqcrypto.org/conferences.html>>
- [15] BERNSTEIN, Daniel J. and LANGE, Tanja, 2017. *Introduction to post-quantum cryptography* [online]. September 2017. Vol. 549, no. 7671, p. 188–194. DOI 10.1038/nature23461. [cit. 2020-10-20]. Available from: <https://www.nature.com/articles/nature23461?error=cookies_not_supported&code=f64d4284-f6c1-4a1a-9698-9efc62562bf5>
- [16] BECKER, Georg, [no date]. *Merkle Signature Schemes, Merkle Trees and Their Cryptanalysis* [online]. [cit. 2020-10-21] Available from: <<https://www.emsec>.

- ruhr-uni-bochum.de/media/crypto/attachments/files/2011/04/becker_1.pdf>
- [17] UNRUH, Dominique, 2017. *Collapsing sponges: Post-quantum security of the sponge construction* [online]. [cit. 2020-10-21]. Available from: <<https://eprint.iacr.org/2017/282.pdf>>
- [18] DONG, P. CH. *Lattice Based Cryptography for Beginners*. Division of General Studies, UNIST. [cit. 2020-10-21]. Available from: <<https://eprint.iacr.org/2015/938.pdf>>
- [19] COUROITS, Nicolas a Louis GOUBIM. *Solving Underdefined Systems of Multivariate Quadratic Equations* [online]. France, Switzerland [cit. 2020-10-22]. Available from: <<http://www.goubin.fr/papers/CG02.pdf>>
- [20] HUYNH, Anh, [no date]. *An Introduction to Supersingular Elliptic Curves and Supersingular Primes* [online]. [cit. 2020-10-22]. Available from: <https://wstein.org/edu/2011/581g/final/anh-supersingular_elliptic_curves.pdf>
- [21] SARAH.HENDERSON@NIST.GOV, 2020. *NIST's Post-Quantum Cryptography Program Enters 'Selection Round.'* NIST [online]. 22 July 2020. [cit. 2020-10-24]. Available from: <<https://www.nist.gov/news-events/news/2020/07/nists-post-quantum-cryptography-program-enters-selection-round>>
- [22] KEY, One and CHUNG, 2004. *Goppa Codes* [online]. [cit. 2020-10-26]. Available from: <<https://orion.math.iastate.edu/linglong/Math690F04/Goppa%20codes.pdf>>
- [23] MCELIECE, J.R. *A public key cryptosystem based on algebraic coding theory* [online]. 1978 [cit. 2020-10-26]. Available from: <https://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF> Communications Systems Research Section
- [24] CHO, Joo, SZYRKOWIEC, Thomas and GRIESSER, Helmut. *Quantum Key Distribution as a Service* [online]. [cit. 2020-11-18]. Available from: <<http://2017.qcrypt.net/wp-content/uploads/2017/09/We476.pdf>>
- [25] BALDI, Marco, SANTINI, Paolo and CHIARALUCE, Franco, 2016. *Soft McEliece: MDPC code-based McEliece cryptosystems with very compact keys through real-valued intentional errors* [online]. [cit. 2020-10-26]. Available from: <<https://arxiv.org/pdf/1606.01040.pdf>>

- [26] HOFFSTEIN, J. *An Introduction to Mathematical Cryptography*. Springer Publishing Company, Incorporated, 2008. ISBN 9780387779935 [cit. 2020-10-27]
- [27] NEJATOLLAHI, Hamid, DUTT, Nikil, RAYR, Sandip, REGAZZONI, Francesco and others (2019). *Post-Quantum Lattice-Based Cryptography Implementations: A Survey*. ACM Computing Surveys. 51. 1-41. 10.1145/3292548.
- [28] ABDOLMALEKI, B. *A Report on Learning With Errors over Rings*. University of Tartu [online], 2016. [cit. 2020-10-27]. Available from: <https://courses.cs.ut.ee/MTAT.07.022/2016_fall/uploads/Main/behzad-report-f16.pdf>
- [29] JOUX, Antoine a Eliane JAULMES. *A Chosen-Ciphertext Attack against NTRU* [online]. France [cit. 2020-10-27]. Available from: <<https://pdfs.semanticscholar.org/a0d1/837e68e3bebc611c60710e667d84cf9ffefb.pdf>>
- [30] LEI, Xinyu and LIAO, Xiaofeng. *NTRU-KE: A Lattice-based Public Key Exchange Protocol* [online]. [cit. 2020-11-19]. Available from: <<https://eprint.iacr.org/2013/718.pdf>>
- [31] LANGLOIS, Adeline and STEHLÉ, Damien, 2014. *Worst-case to average-case reductions for module lattices*. *Designs, Codes and Cryptography* [online]. 26 February 2014. Vol. 75, no. 3, p. 565–599. [cit. 2020-10-27]. DOI 10.1007/s10623-014-9938-4. Available from: <<https://eprint.iacr.org/2012/090.pdf>>
- [32] BOS, Joppe W, L. DUCAS, EIKE KILTZ, T. LEPOINT, VADIM LYUBASHEVSKY, J. SCHANCK, P. SCHWABE and D. STEHLÉ, 2017. *CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM*. undefined [online]. 2017. [cit. 2020-10-28]]. Available from: <<https://www.semanticscholar.org/paper/CRYSTALS-Kyber%3A-A-CCA-Secure-Module-Lattice-Based-Bos-Ducas/a57342d25e255b75a469a8ebc990cb136c6bf2e1>>
- [33] AVANZI, Roberto, Joppe BOS, Léo DUCAS a Eike KILZ. *CRYSTALS-Kyber: Algorithm Specifications And Supporting Documentation*. 2019 [cit. 2020-10-28] Available from: <<https://cryptojedi.org/papers/kybernist-20171130.pdf>>
- [34] ALWEN, Joël, KRENN, Stephan, PIETRZAK, Krzysztof and WICHES, Daniel. *Learning with Rounding, Revisited New Reduction, Properties and Applications* [online]. [cit. 2020-10-28]. Available from: <<https://eprint.iacr.org/2013/098.pdf>>

- [35] BASSO, Andrea, Josa Maria Bermudo MERA, Jan-Pieter D'ANVERS a Angshuman KARMAKAR. *SABER: Mod-LWR based KEM (Round 3 Submission)* [online]. Belgium, 2020 [cit. 2020-10-28]. Available from: <<https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf>>
- [36] ANVERS, Jan-Pieter, SUJOY, Angshuman, ROY, Sinha and VERCAUTEREN, Frederik. *Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM*[online]. [cit. 2020-11-20]. Available from: <<https://eprint.iacr.org/2018/230.pdf>>
- [37] BERNSTEIN, Daniel, LANGE, Tanja and WANG, Wen, 2019. *Classic McEliece: conservative code-based cryptography* [online]. [cit. 2020-10-28]. Available from: <<https://classic.mceliece.org/nist/mceliece-20201010.pdf>>
- [38] WANG, Yang and WANG, Mingqiang, [no date]. *Module-LWE versus Ring-LWE, Revisited* [online]. [cit. 2020-10-28]. Available from: <<https://eprint.iacr.org/2019/930.pdf>>
- [39] AUGOT, Daniel, BATINA Lejla... *PQCRYPTO Post-Quantum Cryptography for Long-Term Security Initial recommendations of long-term secure post-quantum systems*, 2015. [online]. [cit. 2020-10-29]. Available from: <<https://pqcrypto.eu.org/docs/initial-recommendations.pdf>>
- [40] Kyber, 2017. Pq-crystals.org [online]. [cit. 2020-10-29]. Available from: <<http://pq-crystals.org/kyber/index.shtml>>
- [41] SABER: LWR-based KEM, 2017. Kuleuven.be [online]. [cit. 2020-10-29]. Available from: <<https://www.esat.kuleuven.be/cosic/pqcrypto/saber/performance.html>>
- [42] Classic McEliece: Intro, 2017. Mceliece.org [online]. [cit. 2020-10-29]. Available from: <<https://classic.mceliece.org/index.html>>
- [43] NTRU, 2020. Ntru.org [online]. [cit. 2020-10-29]. Available from: <<https://ntru.org/index.shtml>>
- [44] CHEN, Cong, DANBA, Oussama, HOSTEIN, Jerrey, HÜLSING, Andreas, RIJNEVELD, Joost, SCHANCK, John, SCHWABE, Peter, WHYTE, William and ZHANG, Zhenfei, 2019. *NTRU Algorithm Specifications And Supporting Documentation* [online]. [cit. 2020-10-29]. Available from: <<https://ntru.org/f/ntru-20190330.pdf>>

- [45] SINHA, Sujoy and BASSO, Andrea. *High-speed Instruction-set Coprocessor for Lattice-based Key Encapsulation Mechanism: Saber in Hardware* [online]. [cit. 2020-10-29]. Available from: <<https://eprint.iacr.org/2020/434.pdf>>
- [46] *PQ-CRYSTALS*, 2020. pq-crystals/kyber. GitHub [online]. 27 November 2020. [cit. 2020-10-29]. Available from: <<https://github.com/pq-crystals/kyber>>
- [47] *The Ninja build system*, 2020. Ninja-build.org [online]. [cit. 2020-10-29]. Available from: <https://ninja-build.org/manual.html#ref_headers>
- [48] *Kyber-Resources Web*, 2020 [online]. [cit. 2021-04-24]. Available from <<https://pq-crystals.org/kyber/resources.shtml>>
- [49] NOVOSADOVÁ, Tatiana *Crystals-Kyber TCP Implementation*. GitHub [online] [Accessed 2 May 2021]. Available from: <<https://github.com/tnovosadova/Crystals-Kyber>>

List of Abbreviations

AES	Advanced Encryption Standard
ASIC	Australian Securities and Investments Commission
CPU	Central Processing Unit
DDR4	Double Data Rate 4
DPKE	Deterministic Public Key Encryption
DSA	Digital Signature Algorithm
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
GUI	Graphical User Interface
IDE	Integrated Development Environment
IPv4	Internet Protocol Version 4
KEM	Key Encapsulation Mechanism
LWE	Learning With Errors
MLWE	Module Learning With Errors
NASA	Nation Aeronautics and Space Administration
NIST	National Institute for Standards and Technology
NMR	Nuclear Magnetic Resonance
NTT	4Number Theoretic Transform Multiplication Algorithm
QKD	Quantum Key Distribution
RLWE	Ring Learning With Errors
RSA	Rivest–Shamir–Adleman
SDRAM	Synchronous Dynamic Random-Access Memory

SHA	Secure Hash Algorithm
SIVP	Shortest Independent Vector Problem
SVP	Shortest Vector Problem
TCP	Transmission Control Protocol

List of Appendices

A	Installation Guide	63
A.1	Application for macOS	63
A.2	Application for Ubuntu	64
A.3	Application for Other Operating Systems	65
B	User Manual	66
C	How to Create a New Executable	67
D	Shell Scripts	70
E	Contents of the Digital Attachment	73

A Installation Guide

No special installation process is required. To make the application work, start by downloading the entire package available on GitHub [49]. This package contains the source code, as well as already created executable applications.

A.1 Application for macOS

The applications in the "Executables" folder are runnable on macOS Big Sur 11.1 and higher. When attempting to run the application, a pop-up like shown in Figure A.1 might appear. This is a security precaution of the operating system, and can be easily solved.

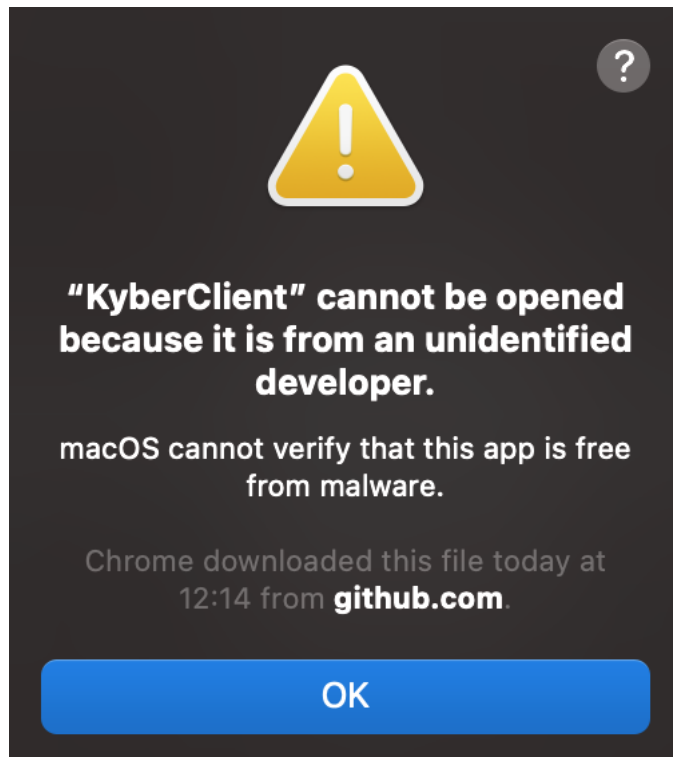


Fig. A.1: Pop-up while opening the downloaded app on macOS.

To solve the above mentioned problem, simply go to **System Preferences** → **Security & Privacy** → **General**. At the bottom of this window, click the lock to allow changes to the settings and click **Open Anyway** (as shown in the Figure A.2). The same process might be required for all of the runnable applications (KyberServer, K2,K3 and K4 in the Server and the Client folder). When the computer trusts all the necessary applications, it is possible to proceed with the usage as described in the following chapter B.

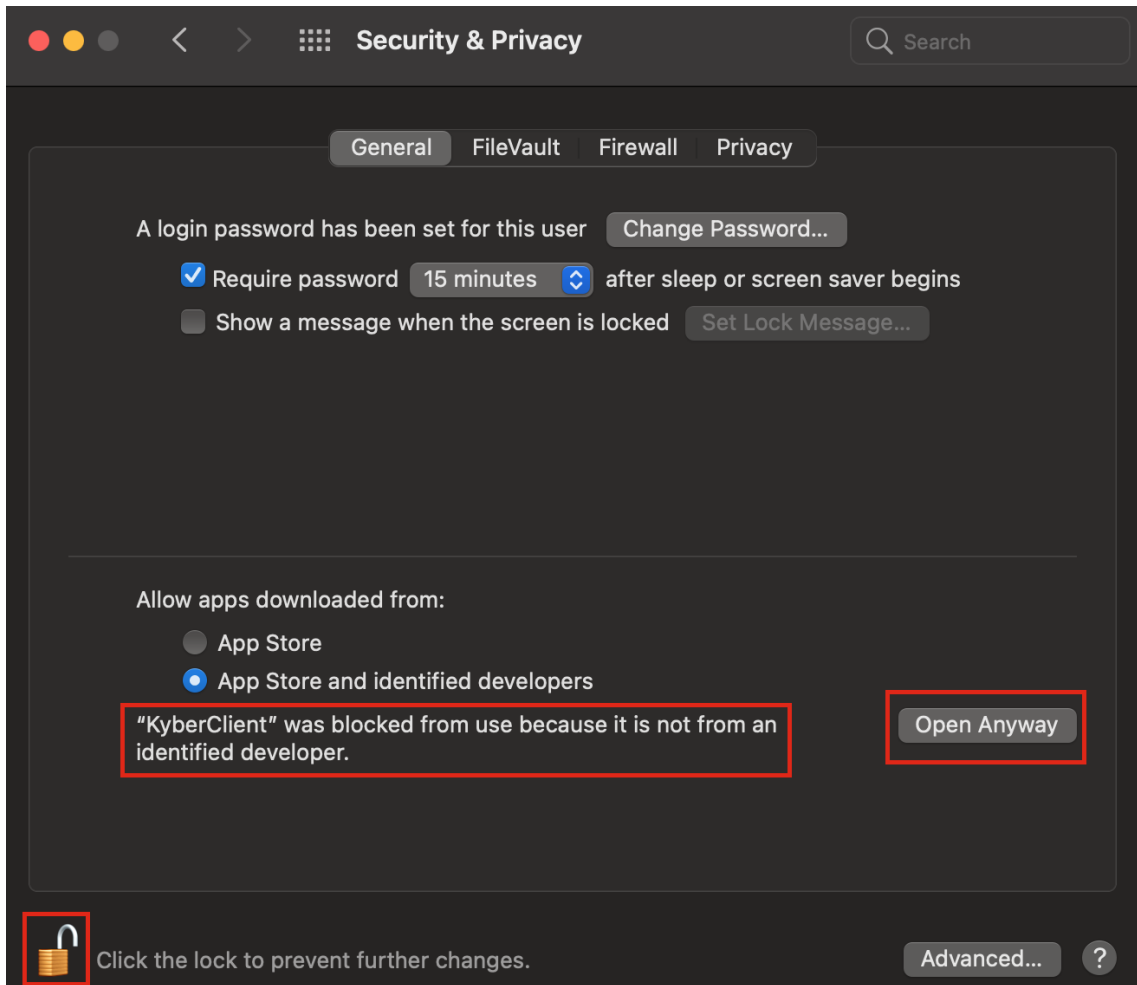


Fig. A.2: Security & Privacy settings to allow the application to run.

A.2 Application for Ubuntu

The package on GitHub [49] also contains applications created for Ubuntu 20.04.2. When trying to run the downloaded application, terminal might say that the permission to run this application is denied (as shown in Listing A.1).

```

1 ubuntu@ubuntu2004:~/Downloads/Crystals-Kyber-main/executables_ubuntu$
  ./KyberServer
2 bash: ./KyberServer: Permission denied

```

Listing A.1: Possible message after trying to run Crystals-Kyber.

To get the permission to run the application, it is necessary to use the following command A.2. The same process needs to be done for all of the executables (KyberClient, K2,K3 and K4 in the Server and the Client folder).

```

1 chmod u+x KyberServer

```

Listing A.2: Command to get the permission to run the application.

After getting the permission to run the necessary applications , it is possible to proceed with the use of the application as described in the chapter B.

A.3 Application for Other Operating Systems

If there is a necessity for the application to be ran on a different operating system, new executables need to be created. The process of doing so is described in the chapter C.

B User Manual

The use of the application is identical for all operating systems.

The first step is to open a terminal window for a Server or a Client and direct themselves to a directory that contains the executable files. If the application is executed on one computer, it is necessary to open two separate terminal windows.

To run the application, type `./KyberServer` and `./KyberClient`. Doing so prompts an explanation and options of running the application B.1.

```
Tatianas-MacBook-Pro:Executables tnovosadova$ ./KyberServer
usage: ./KyberServer options

This is an application for establishing a shared key based on Crystals-Kyber protocol. It is
necessary to RUN THE SERVER FIRST. Parameter for security level needs to match on both sides.

Example of a command for Server: ./KyberServer -l 1024

OPTIONS:
  -h      Show this message
  -l      Security level, can be 512, 768 or 1024

Tatianas-MacBook-Pro:Executables tnovosadova$ ./KyberClient
usage: ./KyberClient options

This is an application for establishing a shared key based on Crystals-Kyber protocol. It
is necessary to RUN THE SERVER FIRST. Parameter for security level needs to match on both
sides.

Example of a command for Client: ./KyberClient -l 1024 -n 100 -a 127.0.0.1

OPTIONS:
  -h      Show this message
  -l      Security level, can be 512, 768 or 1024
  -n      Number of Cycles to be executed, has to be > 0
  -a      IPv4 of Alice (Server)
```

Fig. B.1: Terminal prompt after trying to run the application.

For both, `KyberServer` and `KyberClient` there is an option to choose from different parameters. Parameter `-l` specifies the desired level of security, where `Kyber-512` aims at security roughly equivalent to `AES-128`, `Kyber-768` aims at security roughly equivalent to `AES-192`, and `Kyber-1024` aims at security roughly equivalent to `AES-256` [33]. Parameter `-n` sets the number of key-establishment cycles (which can be any positive integer), and parameter `-a` specifies the IPv4 address of the Server (Alice). When using the application on one device, the address is a local host address - `127.0.0.1`.

Keep in mind to run the Server first!

From here, the entire key-establishment process is automatic. The output is a successfully generated shared key, along with additional information about the time that was needed to execute the command.

C How to Create a New Executable

The project attached to this thesis contains already generated cmake build and necessary files. To create a new executable for a different operational system, these files will need to be disregarded and created from scratch. The process below is using CLion, because this IDE offers the simplest solution.

The first step is to create a new project in CLion. In the left column, make sure to select the C Executable option and select a place to save your new project. In this case, the name of the project is labeled "example". To keep the integrity of the project, it is recommended to use Alice and Bob as a new folder names.

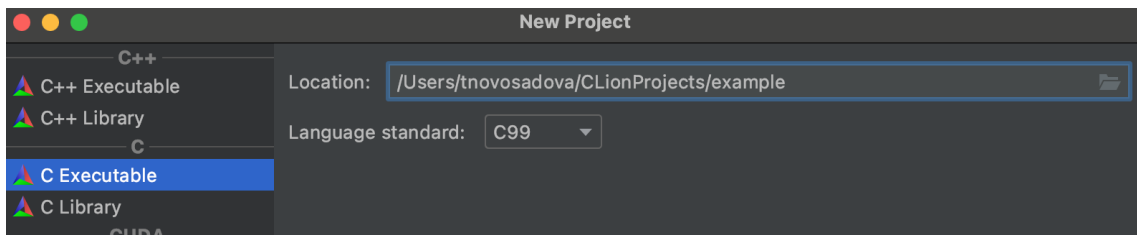


Fig. C.1: Settings for the new project.

Upon creation of the project a cmake-build-debug folder is created. Do not change anything in this folder, everything there is automatically matched to the operational system ran on the computer.

In the next step, the highlighted contents from the Figure C.2 need to be copied into the newly created project. Copy and paste these files into the "example" folder. Accept the pop-up that appears about overwriting the existing main function (Figure C.3).

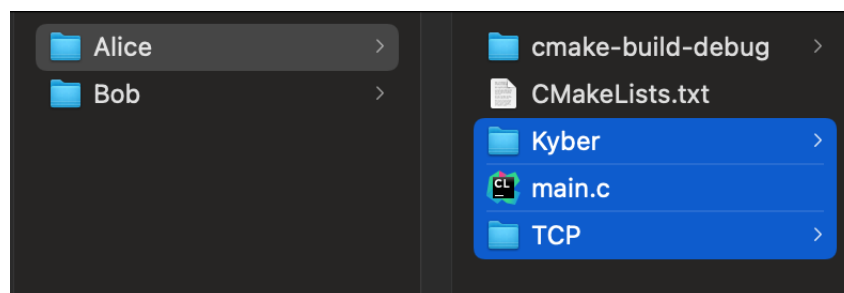


Fig. C.2: Contents of the attached project.

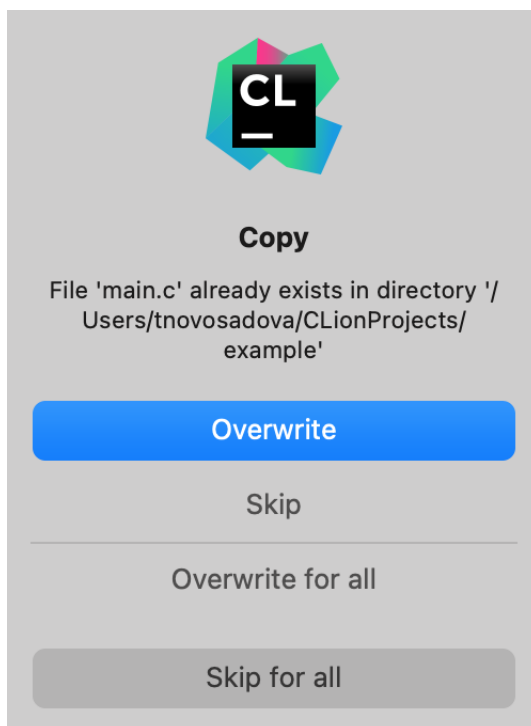


Fig. C.3: Pop-up question to confirm.

The CMakeLists.txt now probably looks like C.1. The line containing `add_executables` needs to be edited. The brackets should contain names of all the files in the project. For example the Listing C.2 is how the line would be changed in CMakeLists.txt for Alice. Same would have to be done for Bob, changing the "Server" parts to Client.

```

1 cmake_minimum_required(VERSION 3.17)
2 project(example C)
3
4 set(CMAKE_C_STANDARD 99)
5
6 add_executable(example main.c)

```

Listing C.1: CMakeLists.txt to be edited.

```

1 add_executable(example main.c Kyber/params.h TCP/Server.c TCP/Server.h
  Kyber/aes256ctr.c Kyber/aes256ctr.h Kyber/cbd.c Kyber/cbd.h Kyber/
  fips202.c Kyber/fips202.h Kyber/indcpa.c Kyber/indcpa.h Kyber/kem.c
  Kyber/kem.h Kyber/ntt.c Kyber/ntt.h Kyber/poly.c Kyber/poly.h
  Kyber/polyvec.c Kyber/polyvec.h Kyber/randombytes.c Kyber/
  randombytes.h Kyber/reduce.c Kyber/reduce.h Kyber/symmetric.h Kyber/
  symmetric-aes.c Kyber/symmetric-shake.c Kyber/verify.c Kyber/
  verify.h Kyber/kex.c Kyber/kex.h)

```

Listing C.2: Changes in CMakeLists.txt for Alice.

After making these changes, it should be possible to run the project without an error, which is how the executables are generated.

The last step is to open the `Kyber/params.h` file. This file contains the definition of the level of security. This is defined as shown in the Listing below (C.3). Number 2 sets the type of the application to Kyber-512, number 3 to Kyber-768 and number 4 to Kyber-1024.

```
1 #define KYBER_K 4
```

Listing C.3: Level of security in `params.h`.

After creating an executable for one level of security, `KYBER_K` value needs to be changed to generate an executable for a different level of security.

Generated executables need to be renamed and located into the right directory, as shown below in Figure C.4.

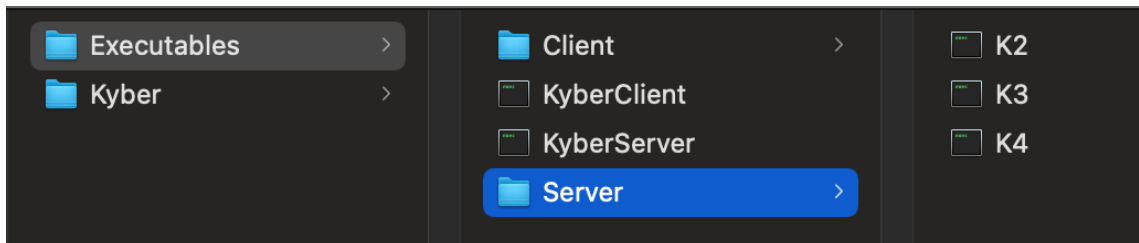


Fig. C.4: Executables directory.

`KyberClient` and `KyberServer` are both shell scripts, that were in this case turned into an executable. It is possible to open them as a plain text if necessary to change something, or copy them from the listings in the Attachment D.

D Shell Scripts

This attachment contains the listings of shell scripts used to execute the CRYSTALS-KYBER applications.

```
1 #!/bin/bash
2 # Argument = -l securitylevel
3
4 usage ()
5 {
6 cat << EOF
7 usage: $0 options
8
9 This is an application for establishing a shared key based on Crystals-
  Kyber protocol. It is necessary to RUN THE SERVER FIRST. Parameter
  for security level needs to match on both sides.
10
11 Example of a command for Server: ./KyberServer -l 1024
12
13
14 OPTIONS:
15     -h      Show this message
16     -l      Security level, can be 512, 768 or 1024
17 EOF
18 }
19
20 choice=
21 while getopts hl: OPTION
22 do
23     case "$OPTION" in
24         h)
25             usage
26             exit 1
27         ;;
28         l)
29             choice=$OPTARG
30         ;;
31     esac
32 done
33
34 if [ -z $choice ]
35 then
36     usage
37     exit 1
38 fi
39
40 if [ $choice == 512 ]
```

```

41 then
42     echo You choose Kyber512
43     ./Server/K2
44 fi
45
46 if [ $choice == 768 ]
47 then
48     echo You choose Kyber768
49     ./Server/K3
50 fi
51
52 if [ $choice == 1024 ]
53 then
54     echo You choose Kyber1024
55     ./Server/K4
56 fi

```

Listing D.1: Shell script KyberServer.

```

1 #!/bin/bash
2 # Argument = -l securitylevel
3
4 usage ()
5 {
6 cat << EOF
7 usage: $0 options
8
9 This is an application for establishing a shared key based on Crystals-
   Kyber protocol. It is necessary to RUN THE SERVER FIRST. Parameter
   for security level needs to match on both sides.
10
11 Example of a command for Client: ./KyberClient -l 1024 -n 100 -a
   127.0.0.1
12
13
14 OPTIONS:
15     -h      Show this message
16     -l      Security level, can be 512, 768 or 1024
17     -n      Number of Cycles to be executed, has to be > 0
18     -a      IPv4 of Alice (Server)
19 EOF
20 }
21
22 choice=
23 number=
24 ip=
25 while getopts hl:n:a: OPTION
26 do

```



```

27     case "$OPTION" in
28         h)
29             usage
30             exit 1
31         ;;
32         l)
33             choice=$OPTARG
34             ;;
35     n)
36         number=$OPTARG
37         ;;
38     a)
39         ip=$OPTARG
40         ;;
41     esac
42 done
43
44 if [ -z $choice ] | [ -z $number ] | [ -z $ip ]
45 then
46     usage
47     exit 1
48 fi
49
50 if [ $choice == 512 ]
51 then
52     echo You choose Kyber512
53     ./Client/K2 $number $ip
54 fi
55
56 if [ $choice == 768 ]
57 then
58     echo You choose Kyber768
59     ./Client/K3 $number $ip
60 fi
61
62 if [ $choice == 1024 ]
63 then
64     echo You choose Kyber1024
65     ./Client/K4 $number $ip
66 fi

```

Listing D.2: Shell script KyberClient.

E Contents of the Digital Attachment

The digital part attached to this thesis contains the final applications that execute the key-establishment process, as well as the source codes used to create these executables. All of the files listed below are also available on GitHub [49].

As mentioned before, the original executables were created for macOS Big Sur 11.1 and higher. Additional applications were created on Ubuntu 20.04.2.

```
/.....root directory
├── Executables_macOS ..... executable files for Crystals-Kyber
│   ├── KyberClient ..... executable for Client's(Bob's) side
│   ├── KyberServer ..... executable for Servers's(Alices's) side
│   ├── Client
│   │   ├── K2 ..... executable for Kyber 512 for Client
│   │   ├── K3 ..... executable for Kyber 768 for Client
│   │   └── K4 ..... executable for Kyber 1024 for Client
│   └── Server
│       ├── K2 ..... executable for Kyber 512 for Server
│       ├── K3 ..... executable for Kyber 768 for Server
│       └── K4 ..... executable for Kyber 1024 for Server
├── Kyber...contains source code for Crystals-Kyber key establishment through TCP
│   ├── Alice
│   │   ├── cmake-build.debug ... contains cmake-files that help create an executable
│   │   ├── CMakeLists.txt ..... executables are build based on contents of this file
│   │   ├── main.c ..... controls the functionality for Alice
│   │   ├── Kyber .... contains source code for key-establishment using Crystals-Kyber
│   │   └── TCP ..... contains source code for TCP communication of a Server
│   └── Bob
│       ├── cmake-build.debug ... contains cmake-files that help create an executable
│       ├── CMakeLists.txt ..... executables are build based on contents of this file
│       ├── main.c ..... controls the functionality for Bob
│       ├── Kyber .... contains source code for key-establishment using Crystals-Kyber
│       └── TCP ..... contains source code for TCP communication of a Client
└── ubuntu_Kyber ...contains source code adjusted to create executables for Ubuntu
    ├── kyber_ubuntu_code .....same directory structure as in Kyber
    └── executables_ubuntu .....same directory structure as in Executables_macOS
```