

Przemysław Samsel

nr albumu: 39427

kierunek studiów: Informatyka

specjalność: Systemy komputerowe i oprogramowanie

forma studiów: *stacjonarne*

Zastosowanie post-kwantowych protokołów ustalania klucza w aplikacjach wymagających wysokiego bezpieczeństwa

Application of post-quantum key protocols in applications requiring high security level

praca dyplomowa inżynierska

napisana pod kierunkiem:

dr hab. inż. Tomasza Hyli

Katedra Inżynierii Oprogramowania

Data wydania tematu pracy: 31.01.2020

Data dopuszczenia pracy do egzaminu:
(uzupełnia pisemnie Dziekanat)

Szczecin, 2019

OŚWIADCZENIE AUTORA PRACY DYPLOMOWEJ

Oświadczam, że praca dyplomowa inżynierska/magisterska (podać rodzaj pracy) pn.

Application of post-quantum key protocols in applications requiring high security level

.....
(temat pracy dyplomowej)

napisana pod kierunkiem: dr hab. inż. Tomasza Hyli

.....
(tytuł lub stopień naukowy imię i nazwisko opiekuna pracy)

jest w całości moim samodzielnym autorskim opracowaniem sporządzonym przy wykorzystaniu wykazanej w pracy literatury przedmiotu i materiałów źródłowych.

Złożona w dziekanacie **Wydziału Informatyki**

(wydział)

treść mojej pracy dyplomowej w formie elektronicznej jest zgodna z treścią w formie pisemnej/pisemnej i graficznej*.

Oświadczam ponadto, że złożona w dziekanacie praca dyplomowa ani jej fragmenty nie były wcześniej przedmiotem procedur procesu dyplomowania związanych z uzyskaniem tytułu zawodowego w uczelniach wyższych.

.....
podpis dyplomanta

Szczecin, dn. 31.01.2020

* niepotrzebne skreślić

Streszczenie

Celem niniejszej pracy jest analiza struktury oraz powiązanych matematycznych problemów dwóch popularnych post-kwantowych protokołów wymiany klucza – FrodoKEM oraz SIDH. Pierwszy z wymienionych został zaimplementowany w języku Python, a sama implementacja była przedmiotem porównania oraz analizy pod kątem szybkości działania oraz bezpieczeństwa, z istniejącą implementacją stworzoną przez Microsoft w C. Aby przedstawić rzeczywisty przykład zastosowania tych algorytmów, przedmiotem implementacji była także aplikacja demonstracyjna w formie czatu internetowego – wykorzystująca albo niezabezpieczone formy komunikacji, albo szyfrowanie przy użyciu popularnego szyfru symetrycznego do którego klucze zostały dostarczone przy użyciu FrodoKEM.

Abstract

Main subject of this work is analysis of structure and related mathematical problems of two post-quantum key-exchange protocols – FrodoKEM and SIDH. The former was also implemented using Python, and this implementation was later a subject of comparison and analysis of efficiency and security with existing implementation in C by Microsoft. To demonstrate real world scenario, demonstration application was developed in a form of simple client-server chat – using either insecure communication methods, or encrypted communication using popular symmetric cipher with keys generated and exchanged by FrodoKEM.

Table of Contents

1. Introduction.....	5
2. Post-quantum cryptography	6
2.1. Background.....	6
2.1.1. Lattices.....	6
2.1.2. Elliptic Curves	7
2.2. Cryptographic systems.....	9
2.2.1. Symmetric cryptography.....	10
2.2.2. Asymmetric cryptography.....	11
2.2.3. Key exchange protocols.....	13
2.3. Quantum computing versus classical cryptography.....	15
2.4. FrodoKEM.....	17
2.4.1. Lattices and vector spaces.....	18
2.4.1.1. Shortest Vector Problem (SVP)	19
2.4.1.2. Learning With Errors (LWE) Problem	21
2.4.2. Algorithm structure.....	22
2.4.3. Motivation for security.....	28
2.5. Supersingular Isogeny Diffie-Hellman (SIDH)	29
2.5.1. Morphisms	30
2.5.2. Isogenies	31
2.5.2.1. Problems related to finding isogenies	32
2.5.3. Motivation for security.....	32
2.5.4. Algorithm structure.....	33
2.5. Python 3.5.x	34
3. Method section.....	35
3.5. Implementation of FrodoKEM in Python	35
3.5.1. Comparison with Microsoft's implementation	37
3.5.2. Fail rate of Python implementation.....	39
3.5. Demonstration application	40
3.6. Existing libraries	44
3.7.1. SIDH implementation in Java by DeFeo Jao and Plut.....	44
3.7.2. FrodoKEM implementation in plain C by Microsoft.....	45
3.7.3. SIDH implementation in plain C by Microsoft.....	47
4. Analysis.....	47

4.1.	Efficiency tests	48
4.2.	Security tests	52
5.	Summary	55
6.	Table of figures & tables.....	56
7.	References	57

1. Introduction

Quantum computing is one of the biggest modern problems that is looming large in many scientists' minds, both in terms of their instability and expected efficiency in comparison to classic computers. The latter property of quantum computers is imminently going to change perspective for cybersecurity as we know it today – simply because common cryptographic algorithms will become either completely insecure or will need improvements in structure (Stubbs, 2018). The so-called "cryptocalypse" has a few mathematical causes. Mainly, the majority of current cryptographic algorithms are based on various difficult math problems i.e. factorizing large integers or discrete logarithms modulo n . However, recent studies have shown that these problems can be easily broken using "*parallel*" computing power of quantum computers. This caused many scientists to start developing new cryptographic systems based on problems which will be tough to compute even for awaited quantum computers. Examples of these problems are Learning With Errors (LWE) based on algebraic structures called lattices, which is a problem of solving system of equations with a probabilistic element, and the other example is constructing isogenies between two elliptic curves over a finite field (NISTIR 8105, 2016).

Based on hardness of mentioned mathematical problems, main objective of this dissertation is a presentation and analysis of two post-quantum cryptographic candidates' applications in terms of their efficiency and offered security. Dissertation begin with short introduction to post-quantum cryptography and basic mathematical primitives – lattices and elliptic curves – on which (later) discussed algorithms are based. In order to bring the reader a broader perspective, first chapters discuss a fundamental understanding of basic concepts related to cryptography – such as cryptographic system and criteria that allow a proper categorization of cryptographic systems. The focus is placed on key exchange protocols, which are the main subject of this dissertation. Last part of first chapter consists of discussion about certain characteristics of quantum computing, latest breakthroughs in this branch of science, and their probable impact on classical cryptography. Lastly, most important categories of post-quantum key exchange protocols designed to run on classical computers will also be shortly introduced.

In the next chapters, FrodoKEM key exchange algorithm is shortly introduced, followed by a brief explanation of its mathematical background as well as related problems. Then the thesis will focus on sections that cover in-depth analysis of its structure, eventually moving to efficiency tests. Its existing implementations, e.g. by Microsoft, will be the subject of general analysis and comparison between them and own implementation in Python. This dissertation also focuses on general knowledge related to quantum computing, its corollaries on classical cryptography, as well as fundamentals of public key cryptosystems. Later chapters also cover Supersingular Isogeny Diffie-Hellman (SIDH) structure and related problems in a briefer manner. The algorithm itself is not the subject of implementation in this work, however, many mathematical functions in Microsoft's original implementation are common between the two analyzed algorithms. Microsoft implemented both SIDH and FrodoKEM in plain C and a few assembly patches, tailoring each loop to exactly suit their needs, thus providing maximum efficiency. It will be interesting perspective comparing to Python's high-level general functions which provide more understandable construction, shorter code, but the efficiency is never going to be any close to the low-level C implementation (Erdem Alkim, 2019).

2. Post-quantum cryptography

Main purpose of this section is to briefly revise the most important cryptographic concepts that are used in this dissertation. Key-exchange algorithms, which are later analyzed, are asymmetrical cryptosystems that are useful in certain situations, and should not be used in others. There are a few most popular key-exchange algorithms in classical cryptography, which are fundamental for understanding quantum-resistant modifications, as in most cases general scheme remained unchanged, but details of key generation or encapsulation/decapsulation are different.

2.1. Background

The mathematical study – called group theory – of various finite algebraic structures has shown to be of a great use in cryptography. Finiteness plays its obvious part here because of computability. Finite sets of elements under different mathematical operations allow us to define unique structures, and these structures are basis for solving certain mathematical problems (Koblitz, 2007). These problems are in turn, the key element to develop algorithms which secure communication. Their security is defined by hardness of solving mathematical problems.

Behavior of mathematical structures is determined by the type of consisting elements i.e. ring of matrices with dimensions 2×2 over real numbers is an example of an infinite ring. Another example would be ring of even integers $2\mathbb{Z}$. Same mathematical rules apply to both, although elements are of completely different types. The one type that drew cryptologist's attention due to its unique structure is an elliptic curve. This geometric object had allowed scientists to develop encryption algorithms that are far more secure, and more efficient than any of algorithms that were used so far. Another mathematical object which is important in terms of this research is a lattice. A set of points in n -dimensional space with a periodic structure is a brief description of that object, and that's where the name come from. Once drew, it resembles a lattice itself (Peikert, 2016).

In order to understand how these algorithms work one needs to therefore get familiar with those mathematical structures. The following paragraphs will acquaint the reader with their definitions, later moving to mathematical problems, related to construction of quantum secure algorithms, which are based on these structures.

2.1.1. Lattices

According to (Regev O. , 2004), “A lattice L is a discrete additive group generated by **ordered set B (b_1, b_2, \dots, b_n)** of n linearly independent vectors. B is also referred to as a basis of lattice. Notice that these vectors belong to R^m , which is m -dimensional space of real vectors. The n is called *rank of the lattice*, and if it equals length of these vectors m , it is called a *full rank lattice*. Similarly, m is called *its dimension*.” (Regev O. , 2004) A lattice in general must satisfy the following conditions:

(*subgroup*) – it must be closed under addition and subtraction;

(*discrete*) – there is always an $e > 0$, so that two distinct points of lattice would satisfy $\|x - y\| \geq e$;

One example of a lattice is a set of all n -dimensional vectors with integer entries. Set Z^n is also a lattice – because vectors fulfill both mentioned criteria (Micciancio, CSE206A Lattice Algorithms and Applications, 2010). Formal definition of a lattice is presented in *Table 1. DEFINITION1 (LATTICE)*.

Given n linearly independent vectors $(b_1, b_2, \dots, b_n) \in R^m$, the lattice generated by them is defined as:

$$L(b_1, b_2, \dots, b_n) = \{\sum x_i b_i \mid x_i \in Z\} \quad 2.1$$

We could equivalently define B as $m \times n$ - dimensional matrix, where each column contains one vector that has length m , then the definition would be:

$$L(B) = L(b_1, b_2, \dots, b_n) = \{Bx \mid x \in Z^n\} \quad 2.2$$

Table 1. DEFINITION1 (LATTICE)

Again, Regev has presented the following examples of lattices:

“A lattice $Z = L((1))$ is a simplest example of full rank lattice, which has 1 dimension and is of rank 1. A lattice could be generated by set of two vectors: $(1,0)^T$ and $(0,1)^T$ and could also be called Z^2 – lattice of all integer points (see *Figure 1. Examples of lattices: full-rank basis of Z^2 (left) and a not full rank lattice based on*). This also would be a full rank lattice. On the other hand, a lattice $L((2,1)^T)$ is not full, as it has dimension 2 and rank 1 (see *Figure 1. Examples of lattices: full-rank basis of Z^2 (left) and a not full rank lattice based on*).

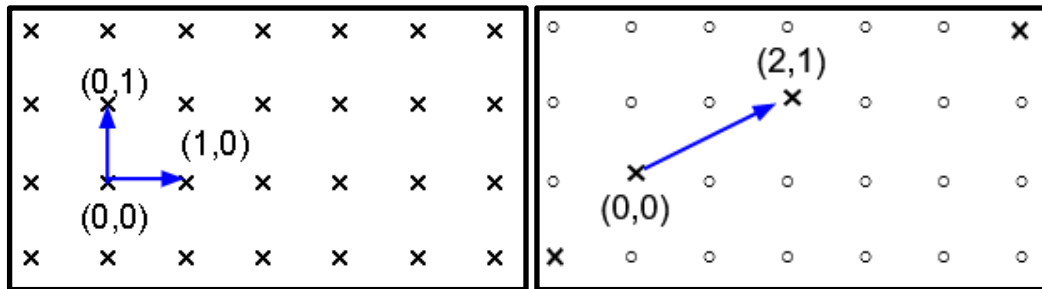


Figure 1. Examples of lattices: full-rank basis of Z^2 (left) and a not full rank lattice based on (Regev O. , 2004)

Lattices are in general simply systematic arrangements of points in Euclidian space. They were present in the nature long before human era in crystals, snowflakes etc. A more day-to-day example could be a stack of fruits packed on the stands in the front of grocery store (Micciancio, CSE206A Lattice Algorithms and Applications, 2010). Even quantum computers' processors have their qubits arranged in a lattice-like structure. Nowadays, lattices are commonly used in mathematics, including computer science. Their usefulness in cryptography is mostly due to hard computational problems they brought to science.

2.1.2. Elliptic Curves

Silverman has provided a following picture of Elliptic Curves in his book (Silverman, 2009):

“Elliptic Curve (EC) is an affine (Cartesian) algebraic set generated by quadratic equations of specific form. They are also called geometric curves of genus one. These curves have their origin in Diophantine equations, which are studied since antient times. Diophantine equations are, in other words, about finding the solution of polynomial equations in integers or rational numbers. EC creates a specific shape when projected onto the plane, hence the name. Cryptography is merely interested in making use of EC that are defined over a finite algebraic structure. This means that calculations on these curves e.g. adding points on the EC over a finite field have finite results (which are very welcome in cryptography).” (Silverman, 2009)

Elliptic Curves have brought much efficiency to public cryptosystems by reducing exchanged key sizes for secure communication, even though the mathematical problem – Elliptic Curve Discrete Logarithm (ECDL) is actually quite similar to this used in “classical” public cryptography (over finite fields).

Elliptic Curve E_1 is a set of solutions $\{(x, y)\}$ over a field k to an equation of the form:	
$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$	2.3
Three most common forms in which ECs are used:	
• <i>Weierstrass Form:</i>	
$y^2 = x^3 + ax + b$	2.4
• <i>Montgomery Form:</i>	
$by^2 = x^3 + ax^2 + x$	2.5
• <i>Legendre form:</i>	
$y^2 = x(x - 1)(x - \Delta)$	2.6

Table 2. DEFINITION2 (ELLIPTIC CURVE) (Silverman, 2009)

One important condition for the use of ECs in cryptography is that, in quantum secure algorithms i.e. SIDH, only *non-singular* curves could be used. This term is not associated with singular points of the curve, but rather a property saying that there is a unique tangent line to the particular EC at its every point. Unfortunately, discrete logarithm problem on a singular curve is not any harder than that on a finite field. It is because group structure on these curves is isomorphic to the multiplicative group of a quadratic extension of a field. Non-singular curves have their discriminant (triangle) not equal to zero (Craig Costello). Figure 2. ECs on a plane a) $y^2 = x^3 - 3x + 3$ ($\Delta = 2160$) b) $y^2 = x^3 + x$ ($\Delta = -64$) c) $y^2 = x^3$ ($\Delta = 0$) presents an example ECs projected on a plane.

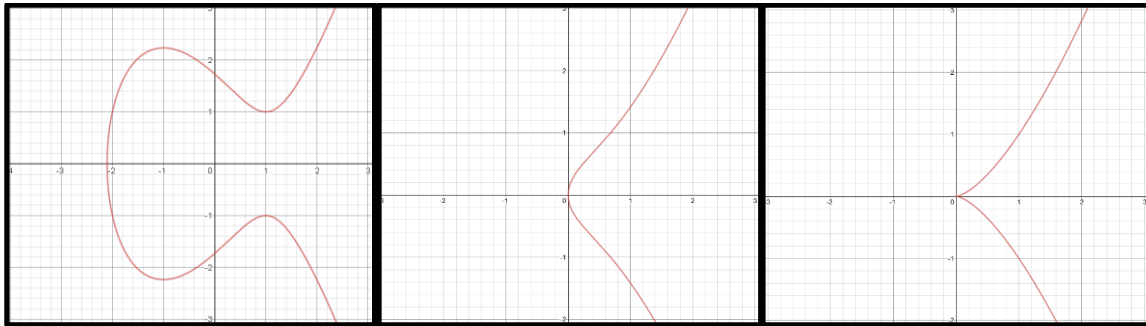


Figure 2. ECs on a plane a) $y^2 = x^3 - 3x + 3$ ($\Delta = 2160$) b) $y^2 = x^3 + x$ ($\Delta = -64$) c) $y^2 = x^3$ ($\Delta = 0$)

Again, according to Silverman's study on Elliptic Curves (Silverman, 2009):

"Elliptic Curve is based on an affine algebraic set of the form V_I , where $I \in \bar{K}[X]$ is ideal, and $\bar{K}[X]$ is a polynomial ring in n variables ($\bar{K}[X_1, \dots, X_n]$). A subset of A^n is associated to every ideal I " (Silverman, 2009):

$$V_I = \{P \in A^n : f(P) = 0 \forall f \in I\} \quad 2.7$$

If V is an algebraic set, the ideal of V is given by

$$I(V) = \{f \in \bar{K}[X] : f(P) = 0 \forall P \in V\} \quad 2.8$$

An algebraic set is defined over K when its ideal $I(V)$ can be generated by polynomials in $\bar{K}[X]$. This is denoted with V/K . When V is defined over K , the set of K -rational points of V is:

$$V(K) = V \cap A^n(K) \quad 2.9$$

Silverman has provided the following examples of algebraic sets (Silverman, 2009):

Example 1 (Silverman, 2009):

- V as the algebraic set in A^2 given by the single equation

$$X^2 + Y^2 = 1 \quad 2.10$$

Where V is defined over K for any field K .

Example 2 (Silverman, 2009):

- The algebraic set

$$V: Y^2 + X^3 + 17 \quad 2.11$$

has many Q -rational points (i.e. $(-2, 3), (\frac{137}{64}, \frac{2651}{512})$). In fact, $V(Q)$ is infinite.

An affine algebraic set V is called an *affine variety* if $I(V)$ is a prime ideal in $\bar{K}[X]$. Note that if V is defined over K , it is not enough to check that $I(V/K)$ is prime in $\bar{K}[X]$ (i.e. ideal $(X_1^2 - 2X_2^2)$ in $Q[X_1, X_2]$).

The study of Elliptic Curves is still developing, and scientists are evaluating their ideas to bring a greater efficiency and security to cryptography. For now, public key cryptosystems based on ECDL problem are secure, but unfortunately not for the longing quantum computers. As these become more and more of a potential threat to modern cryptography, scientists have created algorithms i.e. SIDH discussed in this dissertation, that are based on other problems on Elliptic Curves, which are secure enough even against quantum computers. One of these problems (isogeny problem) is far more abstract than "traditional" ECDL because the subject of operation there is entire curve instead of points, and the operation is creating relations (maps) between curves that have similar structure.

2.2. Cryptographic systems

Providing information security is the main purpose of cryptography. The world nowadays demands secure communication almost everywhere – from business services, through medicine, to personal relationships.

Many public organizations have successfully developed models of secure communication and standards for assessing new cryptographic systems. OSI has developed a security architecture which provides general overview of cryptography's objectives (Stallings):

- *Security attack* – intentional way of compromising organization's valuable digital assets (information);
- *Security mechanism* – process that provides detecting, preventing mechanism or a recovery from a security attack;
- *Security service* – communication service that enhances the security of data storing, processing and communication. Services make use of one or more *security mechanisms* to counter any security attacks;

Cryptographic systems make use of security mechanisms, such as encipherment, and several cryptographic techniques, such as code obfuscation to provide security services, from which most important are Confidentiality, Integrity and Availability (CIA). Cryptographic systems, or cryptosystems, can be simply algorithms (ciphers) to encrypt and decrypt data, and as such are falling into different categories divided by the following criteria (Karbowski, 2013):

- *Type of operations used for transforming plaintext to a ciphertext form* – Based on two fundamental principles – substitution – where each element in the plaintext is mapped into another element, and transposition – where on each iteration the mapping is changing according to a scheme. Essential requirement for all cryptosystems is that no information can be lost. To provide security, all cryptosystems involve multiple stages of both these principle techniques;
- *The number of keys used* – most recognizable category, which divides cryptosystems into two nonconcurrent branches. Symmetric cryptosystems use one key. They are also called as secret-key or conventional encryption. Asymmetric cryptosystems use two keys – public and private. It is also referred to as public key cryptography, and since encryption/decryption operations take more time than for symmetric cryptography, it serves different purposes;
- *The way in which the plaintext is processed* – Two popular examples would be a division into block ciphers and stream ciphers. The former divides a message into blocks and process one block at the time, whereas the latter does not divide a message, but rather process it seamlessly, as it goes.

In the next sections cryptographic systems will be categorized according to second criteria. It is worth to note that cryptography exists since ancient times. One of the first cryptographic devices was Spartan scytale, which embodied transposition cipher. When our civilization became sufficiently developed, and people started to analyze language as a way for communication, Arabic countries developed cryptoanalysis. Since then the battle has started. The battle of creating a way to communicate without being understood by unwanted people versus those “unwanted people” trying to decipher it. Most notorious example of a cryptographic device in history would obviously be Enigma, introduced in XX century by Germany (Singh).

2.2.1. Symmetric cryptography

Symmetric cryptography refers to cryptosystems that require one (secret) key to operate. To provide confidentiality, these cryptosystems require a strong encryption algorithm, that would not allow an opponent to decrypt ciphertext or discover the key, despite intercepting several matching ciphertext-

plaintext pairs or having the knowledge of encryption algorithm. There is also another important requirement saying that keys must be known only to communicating parties, or they must be interchanged securely. Without that, symmetric cryptosystems would unfortunately serve zero security (Stallings). Simplified model of symmetric encryption is presented on *Figure 3*. Simplified model of symmetric cryptosystem .

As can be seen on the figure, message sender uses secret key and encryption algorithm to send encrypted message to the destination through public media. An adversary can effortlessly intercept and try to understand the message. However, until the “code” is broken, it would not make much sense. It is worth to note that security of cryptographic system should not depend on knowledge of encryption algorithm, but

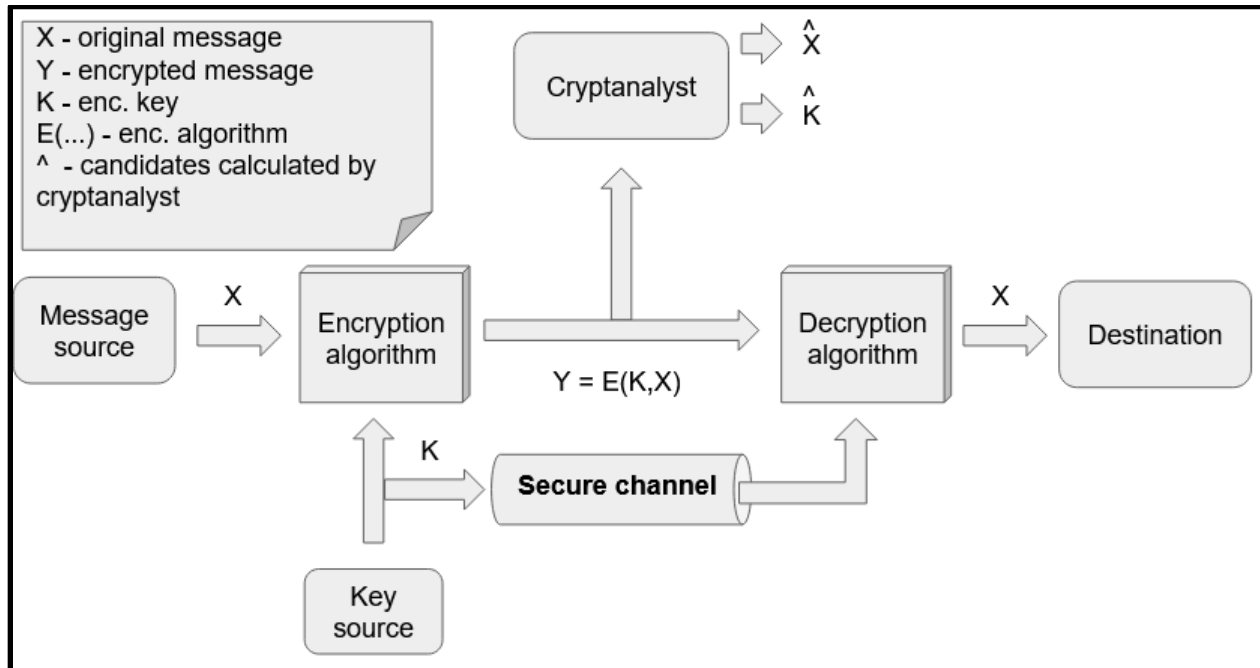


Figure 3. Simplified model of symmetric cryptosystem (Stallings)

rather security of secret keys. As long as the key is secure, communication is as well. Adversary would take one of two approaches – either try cryptanalysis, which relies on the knowledge of algorithm, statistical analysis of language or comparing several ciphertexts and plaintexts or try to brute-force the key. Depending on how large the space of probable keys is, it would take effectively more time to “guess” it. Examples of symmetric cryptosystems would be: Playfair cipher, Hill cipher, Vigenere cipher and even Enigma – from most known historic examples, and DES, AES, IDEA or Blowfish from the computer-era symmetrical cryptography (Karbowski, 2013). Symmetric cryptography is mostly based on several simple operations that are computed interchangeably, rather than some well-known mathematical problems, therefore they are faster and easier to implement either as a software or hardware components. But their biggest caveat is the problem of distribution of the secret keys.

2.2.2. Asymmetric cryptography

To answer weaknesses of symmetric cryptography, mathematicians developed around 1970’s a new category of cryptosystems - asymmetric cryptography. This name refers to ciphers that use a pair of two

keys – public and private key. Having two keys, it is no longer an issue to distribute keys, because public key can be interchanged over public media, and private key is only known to the sender. The tradeoff of this comfort is that asymmetric cryptography takes much longer to compute and usually require larger key sizes. Therefore, it is mostly used to exchange short messages, or secret keys, from where the symmetric cryptosystem takes off. Asymmetric ciphers are mostly based on difficulty of solving some mathematical problems i.e. factorization of large integers or discrete logarithms mod n . Unfortunately, these problems are proven to be no longer to be secure in the era of quantum computers. Later chapters cover that matter in a more descriptive way. For now, the analogous model of asymmetric cryptography looks as in *Figure 4. Simplified model of asymmetric cryptosystem* .

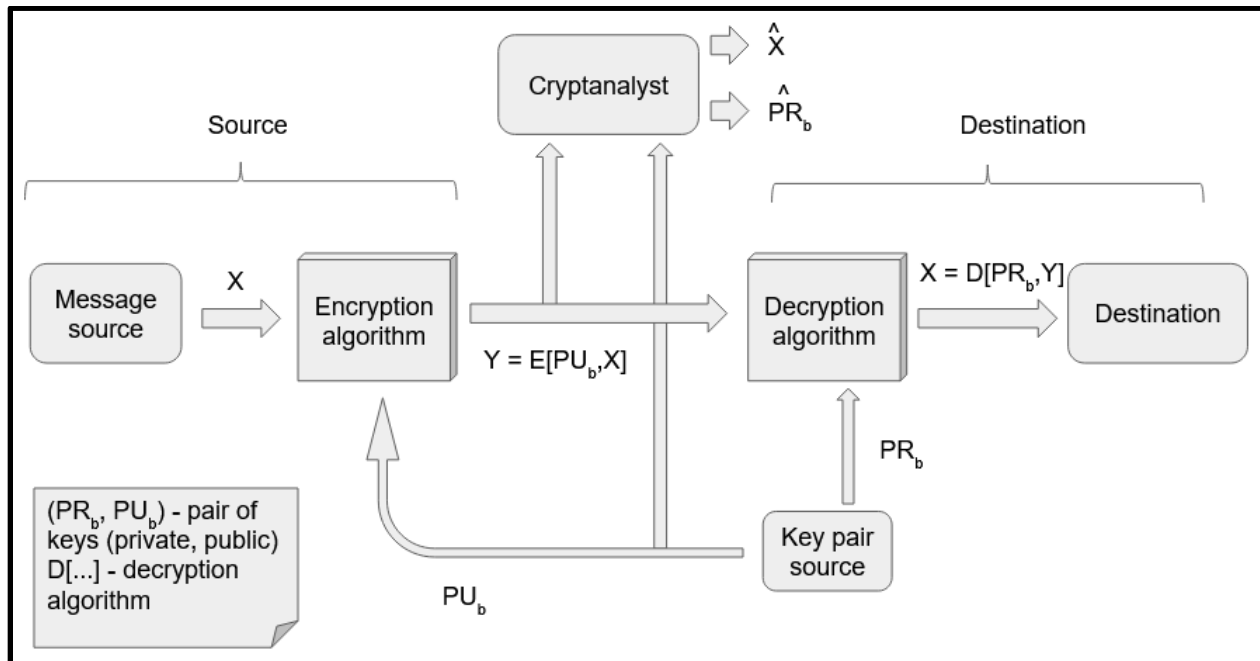


Figure 4. Simplified model of asymmetric cryptosystem (Stallings)

In this model, if parties want to achieve confidentiality, they encrypt messages using each other's public keys. This way, only the subject that owns a paired private key is able to read the message. This solution is far from perfect, since it is inclined to impersonation attacks, such as MiTM (Man in The Middle). These attacks are usually a modification of a following scenario: party A sends its public key, adversary intercepts it and sends own key to party B. When B responds, adversary intercepts also this message and repeats the action towards A. This way, both A and B are unknowingly exchanging "encrypted" messages that adversary has full access to. These kinds of attacks caused cryptographers to develop Public Key Infrastructure (PKI) which i.e. helps to store and exchange authentic certificates containing public keys of communicating parties. Examples of public-key cryptosystems are RSA, El Gamal for encryption as well as i.e. Diffie-Hellman or Elliptic Curve (modification of Diffie Hellman) for key exchange. On the other hand, to achieve authenticity Digital Signature Standard cryptosystem is used. If party A uses own private key to encrypt message, instead of party's B public key, later party B is able to decrypt message using A's public key to confirm that A is the author of the message, assuming that decrypted message makes sense due to agreed convention. General comparison of public-key cryptosystems is presented in the *Table 3*. Comparison of most common public-key cryptosystems .

Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
-----------	-----------------------	-------------------	--------------

RSA	Yes	Yes	Yes
Elliptic Curve	Yes	Yes	Yes
Diffie-Hellman	No	No	Yes
DSS	No	Yes	No

Table 3. Comparison of most common public-key cryptosystems (Stallings)

Public cryptography also uses hash functions in order to authenticate messages. Public cryptosystems and hash functions are used together for creating and signing the digital certificates (Hellman). Hash functions are, on the other way, one-way mathematical functions that transform objects of arbitrary length into objects of a defined length and some special characteristics. Besides cooperation with public cryptography, hash functions are widely used i.e. to determine data integrity in cyber forensics.

2.2.3. Key exchange protocols

Key exchange protocols, or key exchange mechanisms (KEM) are cryptosystems designed specifically for secure exchange of private key over public media. Their structure is generally based on three specific functions:

- *Key generation mechanism* – generates (private, public) keypair;
- *Key encapsulation mechanism* – encrypts private key using public part in order to distribute it securely over the network;
- *Key decapsulation mechanism* – computes shared secret based encrypted private key and some common public parameters.

Structure of key exchange protocols is generally common with other (not designed to exchange keys) asymmetric cryptosystems. They all base their security on some difficult mathematical problems. RSA for example takes advantage of the difficulty to factorize large integers. RSA uses exponentiation modulo some agreed parameter n which is generated using multiplication of two private prime numbers – p and q . RSA key generation creates the pair of public – (n, e) and private (d, n) keys (Stallings). These generated values allow to later encrypt or decrypt shared secret using some specific mathematical functions. For instance, e parameter for public key is calculated as a greatest common divisor (gcd) of e and Euler's totient function of n ($\phi(n)$). The RSA algorithm has been presented in a graphic form in the *Figure 5*. Presentation of RSA algorithm in a graphical form.

This way, security of this algorithm relies on the difficulty of factoring public parameter n in order to recover random integers p and q and finally calculate secret exponent d from a public key (n, e) .

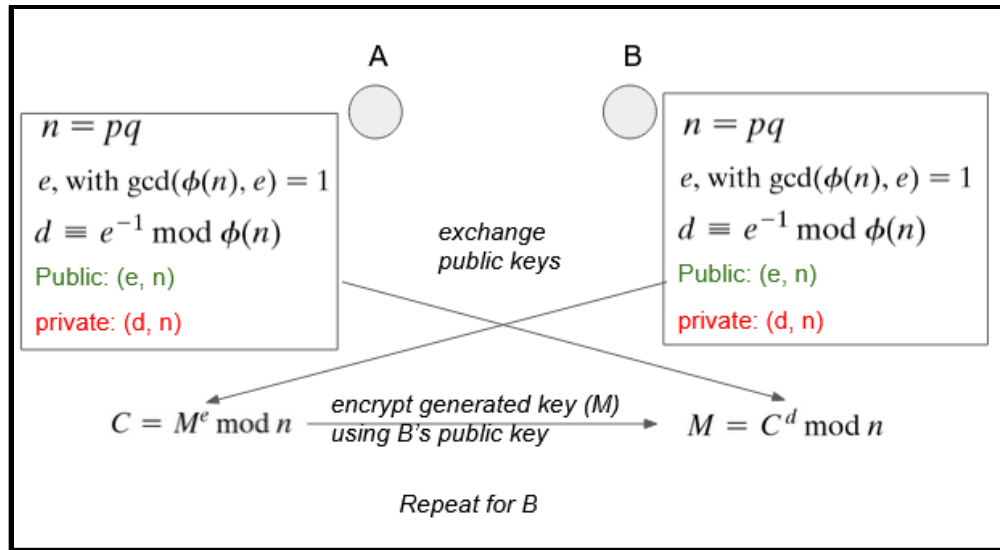


Figure 5. Presentation of RSA algorithm in a graphical form

Another asymmetric algorithm which, conversely is specifically designed for the key exchange is Diffie-Hellman (DH). It is used (as a modified version) in quantum secure algorithm which is in objectives of this dissertation. DH key exchange is based on difficulty of discrete logarithm ($\text{mod } n$) – the problem of finding an exponent k , such that $b^k \pmod{n} = a$, for given parameters b, n . What is important here, is that parameter n , called modulus, should be a prime number, and b should be its prime root, also called the generator. Generator has this important property, saying that when raised to different powers ($\text{mod } n$) the solutions distribute uniformly, or randomly, so it is difficult to establish which specific power gives number a . This requires usually to explore entire space of possible solutions, which very time consuming for large numbers (Cruise).

DH algorithm makes use of discrete logarithm so that both parties agree on some public parameters b and n , then they choose their own private exponent, and raise b to that power. This way, both parties can now send encrypted private keys over public media and after that raise the message received from the other party to the own private exponent's power (Mitchell). They eventually end up doing the same calculations, but in a different order (see Figure 6. Presentation of DH algorithm in a graphical form).

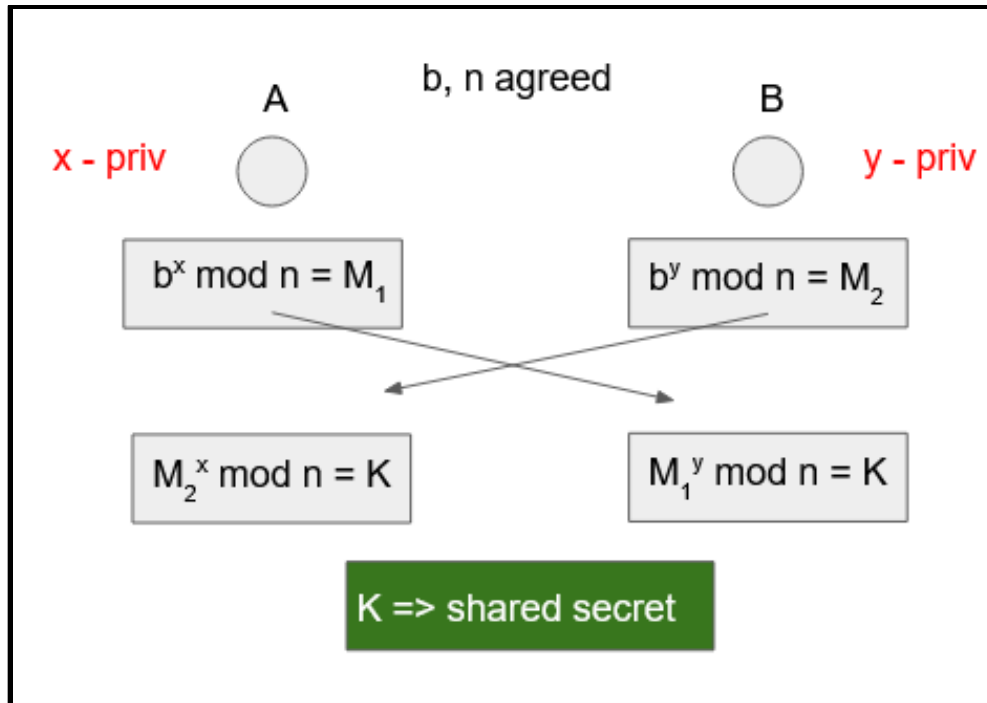


Figure 6. Presentation of DH algorithm in a graphical form

Again, to ensure security, DH algorithm has to use extremely large exponents. Thus, the problem of effective exponentiation modulo some integer arises. There are several approaches to that, some less and some more effective (Singh). Since either way the exponentiation of large integers became troublesome for computer scientists, because of larger and larger key sizes, they eventually end up looking for a more efficient solution. One of them is based on similar problem, called Elliptic Curve Discrete Logarithm Problem (ECDL). Calculations on points on Elliptic Curves allowed to change low-efficient exponentiation to simpler operations and reduce key sizes. However, in the face of quantum computing era, even this solution could not provide sufficient security (Hellman). Therefore, scientists started researching new problems to improve securing communication/key exchanging.

Finally, quantum computing does provide another solution for key exchange – called quantum key distribution. As opposed to classical asymmetric solutions, it is mostly based on physics, rather than mathematical problems. It exploits a fact that observations (or measurements) of quantum state introduces perturbations in that state. These perturbations are detectable by the receiver as noise present in shared secret, making it possible to detect all kinds of impersonation, or MiTM attacks. However, this solution requires an authenticated channel of communication between exchanging parties (Renner, 2014).

2.3. Quantum computing versus classical cryptography

Quantum computers have been of a particular interest for a long time, since their expected computational power is times more powerful than any classical supercomputers currently at dispose. One scientific phenomenon that makes this possible is certainly a superposition of qubits. Qubits are the unit of information that quantum processors use for computation. Qubits are able to store the same information as bits in classical computers, however their superposition makes possible to be both 1 and 0 simultaneously,

whereas classical computers can compute only one combination at the time (Savage, 2019). This gives quantum computers great advantage in terms of computing efficiency, which is also the source of their main caveat – their computations are presently very unstable and error prone. Therefore, it is still a technology in its earliest stage that is a great challenge for scientists and will certainly require time to develop. Conversely, giving all the obstacles that quantum computers have to overcome, there is actually some progress going on. Scientists from Google recently published a paper about their experiment with quantum computer that has 54 qubits onboard. Sycamore, which is the name of the machine, was used to produce enormous range of random numbers and test them to ensure whether their randomness was true. It took about 200 second to finish computations and it also happened that only 53 qubits were eventually able to finish the task. Even though it was trivial, it would have taken 10 000 years the best of classic supercomputers to compute (Frank Arute, 2019). This is one of most important breakthroughs in quantum computing since IBM scientists introduced first quantum machine in 1984.

The implications of quantum computing supremacy (or the era of quantum computing, as it is often referred to) for the classical cryptography are going to break all current public key cryptosystems. Be it either symmetric or asymmetric cryptosystems, they will require either modifications, or will no longer be considered secure. This is thanks to Shor's algorithm originally published in 1994, that has been proven to effectively factorize large integers in no time, as well as to solve Discrete Logarithm Problem modulo n over various groups (Shor., 1994). These two mathematical problems are the foundation of entire public key/asymmetrical cryptography – of algorithms such as RSA or DH. Another research that has led to developing Grover's search algorithm would also make cryptanalysis against symmetric ciphers, or even collision finding for hash algorithms substantially faster, therefore forcing scientists to restructure them to provide sufficient security (NISTIR 8105, 2016). *Table 4.* Summary of quantum computing implications for cybersecurity presents the summary of quantum computing implications for cybersecurity.

Apparently, the concern about quantum computers has been shared over the years around the globe. Governments, biggest corporations have recently started considering the matter seriously, and are currently adapting to the ever-changing environment by financing research of quantum secure cryptography, public organizations are publishing standards, guidelines and numerous other kinds papers about the risk that quantum cryptography would bring.

Cryptographic algorithm	Type	Purpose	Impact from large-scale quantum computer
AES	Symmetric key	Encryption	Larger key sizes needed
SHA-2, SHA-3	-----	Hash functions	Larger output needed
RSA	Public key	Signatures, key establishment	No longer secure
ECDSA, ECDH	Public key	Signatures, key exchange	No longer secure
DSA (Finite Field Cryptography)	Public key	Signatures, key exchange	No longer secure

Table 4. Summary of quantum computing implications for cybersecurity (NISTIR 8105, 2016)

There are in fact four main classes of mathematical primitives that offer security against quantum computing (NSA, 2015):

- *Lattice-based cryptography* – basing on problems such as LWE, lattices are now of a great interest to scientists, leading to developing new approach in cryptography, such as homomorphic encryption – type of encryption that allows operating on ciphertext which once decrypted, match the result of the same operations on the plaintext. Examples of such cryptosystems would be Hoffstein, Pipher and Silverman’s NTRU algorithm; Peikert’s NewHope or FrodoKEM;
- *Code-based cryptography* – McEliece cryptosystem was created in 1978, but since now has not been widely applied. McEliece uses linear codes that are used in several types of error correcting codes;
- *Multivariate polynomial cryptography* – focusing on difficulty of solving systems of multivariate polynomials over finite fields. An example of such algorithm would be Patarin’s HFE⁻ signature scheme;
- *Hash-based signatures* – Merkle’s hash-tree signatures fall into this category. These algorithms however require keeping track of all the messages that have been signed, which, giving also that the total number of signatures that specific algorithm is finite, is a considerable drawback of these branch of quantum secure cryptography;

Recently, there have also been some research by Microsoft that does not fall into any of these categories. Elliptic curves, which have already been used to a great extent in public key cryptography, have evolved into a new form. Instead of operations on specific points on an elliptic curve, Microsoft has proposed to operate on a specific set of elliptic curves. This new approach, called SIDH, is believed to substantially advance research on quantum secure cryptography (Craig Costello). One advantage of abovementioned cryptosystems over classical ones, is that they are believed to provide security both against classical and quantum cryptanalysis. They also come with their price – all these algorithms require either considerably larger key sizes to operate securely or computations take much more cycles. This is very unfortunate considering that quantum algorithms have to be implemented and prove their feasibility long before quantum computers will become reality.

2.4. FrodoKEM

FrodoKEM is a lattice based key exchange mechanism which bases its security on mathematical *learning with errors* (LWE) problem. Public-key encryption scheme called FrodoPKE is the core structure of this algorithm. First cryptographic constructions based on various problems on point lattices in \mathbb{R}^n were conceived back as far as in 1997. Based on the work published in 2005 by Regev, which defined LWE problem and proved the hardness of it considering specific parameters and proposed first version of public encryption scheme based on LWE. The initial work was later improved in a work published in 2011 by Lindner and Peikert (Richard Lindner, 2011). FrodoKEM is considered having IND-CPA (based on FrodoPKE) security level, which is simply a property of cryptosystems that makes adversary unable to distinguish ciphertexts based on decrypted message (Erdem Alkim, 2019). FrodoKEM is also considered IND-CCA secure. These terms will be further discussed in later sections of this chapter.

FrodoKEM has now three different levels of security implemented:

- FrodoKEM-640 – targeting brute-force security of AES-128;
- FrodoKEM-976 – targeting brute-force security of AES-192;
- FrodoKEM-1344 – providing the highest security amongst FrodoKEM versions, matching or exceeding AES-256;

Each of these schemes also have two variants for pseudo randomly generating matrix A with two different functions: AES128 and SHAKE128. First variant is particularly recommended for architectures having AES hardware acceleration. Giving this basic description of FrodoKEM, this chapter is about to familiarize the reader with basic concepts related to this key encryption scheme, as well as demonstrate its algorithmic structure based mostly on official documentation (Alkim, 2019) as well as other sources.

2.4.1. Lattices and vector spaces

Earlier chapters defined the basic definition of a lattice. This section provides an extension for basic concepts already discussed in order to eventually build a broader understanding of lattice mathematics that would allow the reader to delve into a world of lattice-based problems. Lattices in computer science are represented by finite objects called basis matrix B that generates the lattice. The matrix has integer or rational entries. The definition of a lattice is, to recap, was as follows (Goldwasser, 2002):

$$L(B) = L(b_1, b_2, \dots, b_n) = \{Bx \mid x \in \mathbb{Z}^n\} \quad 2.11$$

Which is a set of all the integer linear combinations of the columns of basis B . One important mathematical object based on lattices is span of a lattice. It is, in other words the linear space spanned by lattice's vectors:

$$\text{span}(L(B)) = \text{span}(B) = \{By \mid y \in \mathbb{R}^n\} \quad 2.12$$

Notice the similarity between definitions of a lattice and its span. Micciancio has given a following explanation about a span of a lattice in his paper (Micciancio, Introduction to Lattices, 2010):

“Span is also called a vector space. Its main difference in structure between lattice is that one can combine columns of B with arbitrary real coefficients, while lattice base can only include integer coefficients – resulting in a discrete set of points. This way one can define multiple operations on vectors in lattice. Since vectors b_1, \dots, b_n are linearly independent, arbitrary point y that of the span can be used as a linear combination $y = x_1b_1 + \dots + x_nb_n$ in a unique way. Therefore, y belongs to a lattice if and only if x_1, \dots, x_n belongs to \mathbb{Z} . Base matrix B is also a basis for the $\text{span}(B)$, if it is basis for lattice $L(B)$. However, not every basis for the vector $\text{span}(B)$ is a lattice basis for $L(B)$. For instance, $2B$ is a basis for $\text{span}(B)$ as a linear space, but it is not a basis for $L(B)$ as a lattice because vector b_i that belongs to $L(B)$ is not an integer linear combination of the vectors in $2B$.” (Micciancio, Introduction to Lattices, 2010)

Another object of interest in case of lattices is a fundamental parallelepiped. It is visually represented as a regular tiling of entire $\text{span}(L(B))$ (see Figure 7. Fundamental parallelepiped visual representation . It is commonly used fundamental domain which is centred to the origin of lattice.

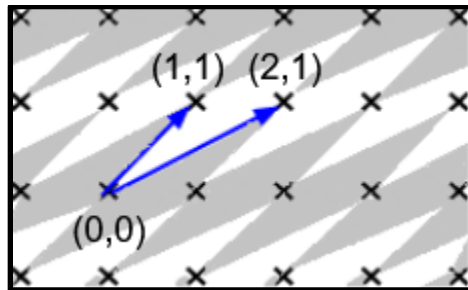


Figure 7. Fundamental parallelepiped visual representation (Regev O. , 2004)

In mathematical sense, its definition is covered by the following equation:

$$P(B) = B[0,1]^n = \{Bx \mid x \in \mathbb{R}^n, \forall_i: 0 \leq x_i < 1\} \quad 2.13$$

Micciancio provided an accurate definition of fundamental parallelepiped (Micciancio, CSE206A Lattice Algorithms and Applications, 2010):

“In other words, fundamental parallelepiped is just a set of points that satisfy the $0 \leq x_i < 1$ equation. Since $P(B)$ is half-open, translates $P(B) + v$ (for any vector v that belongs to the lattice $L(B)$) form a partition of the whole space \mathbb{R}^k .” (Micciancio, CSE206A Lattice Algorithms and Applications, 2010)

The definition of a fundamental parallelepiped allows to introduce the determinant, a fundamental quantity associated to any lattice. This account is from Regev’s paper (Oded Regev, 2004):

“For any lattice $L(B)$ of base B , the determinant is defined as a generalization of the area of a parallelepiped. In the sense, the determinant represents the inverse of the density of lattice points in a space (i.e. number of lattice points on a large region of space should be equal to the volume of lattice divided by the determinant) – the smaller the determinant, the denser the lattice is. The determinant does not depend on the choice of the basis.” (Oded Regev, 2004) Formal definition of the determinant looks as follows (where b_i^* is element of the Gram-Schmidt orthogonalization of B , and will be discussed later):

$$\det(L(B)) = \text{vol}(P(B)) = \prod \|b_i^*\| \quad 2.14$$

Finally, based on Regev’s work (Oded Regev, 2004), few more words about the determinant:

“Last important quality of lattices that should be considered is determining whether two separate bases of lattice B_1, B_2 generate the same lattice ($L(B_1) = L(B_2)$). There are basically two ways – one says that two bases are equivalent if and only if $B_2 = B_1 U$ for some unimodular matrix U . Unimodular matrix is simply a matrix $U \in \mathbb{Z}^{n \times n}$ which determinant $\det(U) = \pm 1$.” (Oded Regev, 2004)

The determinant of a matrix is not associated with determinant of a lattice. The other way says that two lattice bases are equivalent only if one is obtained from one another by the following operations on columns:

1. $b_i \leftarrow b_i + kb_j$ for some $k \in \mathbb{Z}$;
2. $b_i \leftrightarrow b_j$
3. $b_i \leftarrow -b_j$

Simply put, $L(B_1) = L(B_2)$ if any of the columns are transformed by adding another column multiplied by some integer k , subtracted by each other or just changed their order.

2.4.1.1. Shortest Vector Problem (SVP)

The Shortest Vector Problem is either about finding a non-zero-length vector in lattice (decisional version) or finding such a vector which length is at most g times the length of an optimal solution, where the approximation factor g is a function of the lattice dimension.

In order to better understand the problem, a few other definitions associated with lattices must be explained. One important is Gram-Schmidt Orthogonalization/Process. It is a basic procedure widely used in linear algebra that takes as an input set of n linearly independent vectors to output a set of n orthogonal vectors,

by projecting these vectors on the space orthogonal to the span of lattice (Micciancio, CSE206A Lattice Algorithms and Applications, 2010). By definition, Gram-Schmidt orthogonalization is a sequence of vectors $\mathbf{b}_1, \dots, \mathbf{b}_n$ which satisfies:

$$\mathbf{b}_i = \mathbf{b}_i - \sum_{j=1}^{i-1} u_{i,j} \mathbf{b}_j, \text{ where } u_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j \rangle}{\langle \mathbf{b}_j, \mathbf{b}_j \rangle} \quad 2.15$$

Or simply \mathbf{b}_i is the component of \mathbf{b}_i , orthogonal to $\mathbf{b}_1, \dots, \mathbf{b}_{i-1}$. In matrix notation, orthogonalized vectors B^* satisfy $B = B^*T$, where T is an upper triangular matrix with 1's on the diagonal, and the $u_{i,j}$ coefficients at position (j, i) for all $j < i$. Now, the formula for the determinant of a lattice can be written as:

$$\sqrt{\det(B^*TB^*)} = \prod \langle \mathbf{b}_i, \mathbf{b}_i \rangle = \left(\prod \|\mathbf{b}_i\|^2 \right) = \det(L(B))^2 \quad 2.16$$

Some important properties of Gram-Schmidt orthogonalization are:

- For any $i \neq j$ corresponding $\langle \mathbf{b}_i, \mathbf{b}_j \rangle$ equals 0;
- For all $1 \leq i \leq n$, $\text{span}(\mathbf{b}_1, \dots, \mathbf{b}_i) = \text{span}(\mathbf{b}_1, \dots, \mathbf{b}_i)$;
- Vectors $\mathbf{b}_1, \dots, \mathbf{b}_n$ need not to be a basis B of $L(B)$. They are usually not even contained in that lattice;
- What is also important, the order of vectors $\mathbf{b}_1, \dots, \mathbf{b}_i$ matters. They are therefore considered rather a sequence than *set*;

Another basic parameter of a lattice is the length of the shortest nonzero vector in the lattice – successive minimum. This parameter is denoted by λ_i . Length has here a meaning of the Euclidean norm, defined as:

$$\|x\|_2 = \sqrt{\sum x_i^2}, \text{ also denoted as } \|x\| \quad 2.17$$

The definition of λ_i is that it is the smallest r such that the lattice points inside a ball of radius r a space of dimension i . Formal definition looks as follows:

$$\lambda_i = \inf \{ r \mid \dim(\text{span}(L(B) \cap \beta(0, r))) \geq i \} \quad 2.18$$

where $\beta(0, r) = \{x \in \mathbb{R}^m \mid \|x\| \leq r\}$ is the closed ball of radius r around 0 . Important term that helps in defining successive minima are its useful lower bound and upper bound. They are based on Minkowski's theorem; however, these definitions are beyond the scope of this dissertation and therefore will not be further discussed.

Minkowski's first theorem implies the existence of a nonzero vector of length at most $\sqrt{n}(\det(L))^{1/n}$ in any lattice L . However, the theorem does not propose any algorithm to find such short vector. Therefore, the Shortest Vector Problem was created. According to Regev (Regev O. , 2004):

“Given a lattice L , the problem is to find the shortest nonzero lattice point. There are in fact three most common variations of this problem:

- *Search SVP*: given a lattice basis $B \in \mathbb{Z}^{m \times n}$ find $v \in L(B)$ such that $\|v\| = \lambda_1(L(B))$;
- *Optimization SVP*: Given a lattice basis $B \in \mathbb{Z}^{m \times n}$ find $\lambda_1(L(B))$;
- *Decisional SVP*: Given lattice basis $B \in \mathbb{Z}^{m \times n}$ and a rational $r \in \mathbb{Q}$, determine whether $\lambda_1(L(B)) \leq r$ or not;

Two relations among the three variants above are that the decision variant is not harder than the optimization variant, and that the optimization variant is not harder than the search variant. In summary, these three variants are essentially equivalent. Often finding exact solutions is not necessary. Instead, one is usually interested in finding an approximation of it.” (Oded Regev, 2004)

The approximation factor is defined by some parameter $\gamma \geq 1$. Formal definition of the Approximate Shortest Vector Problem (SVP γ) is as follows (Oded Regev, 2004):

Given lattice L , find a nonzero vector $v \in L$ for which $\|v\| \leq \gamma(n) \times \lambda_1(L)$

Finally, the Decisional Approximate SVP – formally called GapSVP γ is defined as (Oded Regev, 2004):

Given basis B of a full-rank lattice L , output a set $S = \{s_i\} \subset L$ of n linearly independent lattice vectors where $\|s_i\| \leq \gamma(n) \times \lambda_1(L)$ for all i

According to Micciancio (Micciancio, CSE206A Lattice Algorithms and Applications, 2010):

“Approximating SVP in the Euclidean norm is *NP-hard* (under randomized reductions) for any constant approximation factor $O(1)$. Therefore, finding a polynomial space algorithm to solve SVP in single exponential time $2^{O(n)}$ is thoroughly researched. All currently known algorithms running in single exponential time make use of exponential space too. Best approach so far is given by enumeration method.” (Micciancio, CSE206A Lattice Algorithms and Applications, 2010)

2.4.1.2. Learning With Errors (LWE) Problem

Learning With Errors is a very distinctive problem in a mathematical sense, because it brings hardness based on the worst-case lattice problems, which the best classical algorithms are capable of solving in exponential time, and better quantum solutions do not exist. Therefore, it is conceived very secure even in the face of quantum computing era. Best classical algorithm was created by Blum, Kalai and Wasserman (Avrim Blum, 2003).

According to Regev (Regev):

“LWE is simply a problem of finding solution s to a set of n linear equations, which are disturbed by some small additive error. If it was not for the random error (noise), the system of equations could successfully be solved in polynomial time using i.e. *Gaussian elimination*. However, introducing the noise in equations amplifies the problem to unmanageable levels for standard solving methods.” (Regev)

To give a more precise definition, there are given a set of n linear approximate equations modulo q , where $n \geq 1$ and $q \geq 2$ are fixed and agreed parameters, and an error probability distribution χ on Z_q is introduced. The $A_{s,\chi}$ on $Z_q^n \times Z_q$ is the probability distribution obtained by the vector $a \in Z_q^n$ chosen randomly, with the error value $e \in Z_q$, according to (Gaussian) error distribution χ . This way, each vector is re-generated, and outputs $(a, \langle a, s \rangle + e)$, where all additions are performed in Z_q . The solving algorithm should, given an arbitrary number of independent samples from $A_{s,\chi}$, output a solution vector s with high probability (Apon, 2012). Note similarity to other lattice related problems, as well as other mathematical structure-based. LWE could be equivalently defined as a problem of decoding random linear codes, or a random bounded distance decoding problem on lattices.

The structure of linear equations for LWE would look as follows (Martin R. Albrecht, 2019):

$$\begin{aligned} a_1 &\leftarrow Z_q^n, & b_1 &= \langle s, a_1 \rangle + e_1 \\ a_2 &\leftarrow Z_q^n, & b_2 &= \langle s, a_2 \rangle + e_2 \\ & & \dots & \\ a_n &\leftarrow Z_q^n, & b_n &= \langle s, a_n \rangle + e_n \end{aligned} \tag{2.19}$$

Where errors are $e_i \in \chi$ Gaussian over Z , param αq

There are, in fact, a few concepts that prove LWE hardness. As mentioned before, best known algorithms for LWE run in exponential time. Because it is a somewhat extended version of another problem, Learning Parity With Noise (LPN) – which is beyond the scope of this work. Again, hardness of LWE is known on the base of other standard lattice-based problems such as GapSVP, or SIVP (Shortest Independent Vectors Problem). All mentioned problems are also hard to approximate within polynomial factors. For polynomial moduli q , the hardness is based on slightly less standard assumptions, namely, GapSVP is hard to approximate given a short lattice basis, and finally all mentioned problems are difficult to approximate using either classical or quantum algorithms (rafael pass, 2010). This variation with polynomial rings over finite fields is a broader problem than LWE and is called Ring Learning With Errors (RLWE).

A naïve way to solve LWE is through a maxim likelihood algorithm. Under certain conditions, it is not difficult to prove that after about $O(n)$, the only assignment to s that satisfies the equations approximately should be the correct one. This leads to a probabilistic algorithm that runs in $2^{O(n \log n)}$ time. A solution created by Blum is based on an idea that follows to find a small set S of equations, among $2^{O(n)}$ equations, such that $\sum S a_i$, say $(1, 0, \dots, 0)$. By summing these equations, the first coordinate of s can be recovered - and similarly for all other equations thus solving entire system (Regev).

2.4.2. Algorithm structure

Main source of intractability, on which the security in FrodoKEM is based upon, are LWE problem and its variations, such as Ring-LWE or Module-LWE - which are on the contrary based on “algebraically structured” lattices; even problems related with the classic NTRU cryptosystem – all of them are related to plain LWE and have a lot of computational aspects that make them more attractive, however this “extra structure” added to the classic LWE causes scientists to stay skeptic about security of algorithms based these extensions. For this reason, the decision was made that FrodoKEM would rely its computations (and therefore security) on rather just the plain LWE problem – given the unpredictable long-term outlook of potential hardness for algebraically structured lattices’ problems, either for classical or quantum cryptography. The top-level structure of FrodoKEM, as well as similar KEMs is based on a collection of functions (Erdem Alkim, 2019) that operate on specific finite key space (K):

- *KeyGeneration()* $\rightarrow (pk, sk)$ – probabilistic algorithm that outputs a pair of public (pk) and secret (sk) keys;
- *Encapsulation*(pk) $\rightarrow (c, ss)$ – probabilistic algorithm that, given public key as an input, outputs an encapsulation/ciphertext (c) and a shared secret (ss) $\in K$;
- *Decapsulation*(c, sk) $\rightarrow ss'$ – deterministic decapsulation algorithm that takes encapsulation and secret key as input and outputs a shared secret (ss') $\in K$.

This shared secret should match ss produced by encapsulation. In other words, KEM is δ -correct if it satisfies:

$$P([ss' \neq ss : (pk, sk) \leftarrow \text{KeyGeneration}(); (c, ss) \leftarrow \text{Encapsulation}(pk); ss' \leftarrow \text{Decapsulation}(c, sk)]) \leq \delta$$

Which basically says that probability of these secrets not matching should be smaller than a certain threshold (δ). Parameters for FrodoKEM are conservatively designed with simplicity in mind, leaving small or no space for errors. Main assumptions are (Erdem Alkim, 2019):

- *Compact base code* – original Microsoft’s implementation in C (for x86 processors) is manifested through a concise, 250 lines of code, which do not change a lot for achieving higher NIST-security levels.
- *Matrix and vector operations* – besides error sampling and symmetric primitives (AES, SHAKE functions used in generating matrix A) main operations are based on simple matrix-vector products, rather than implementing specialized multiplication algorithms which exploit lattices structure;
- *Modular arithmetic* – should use integer modulus $q \leq 2^{16}$, which is always a power of two; this ensures that only single-precision arithmetic is needed, and reductions mod q can be computed at low cost. Modular arithmetic should be therefore easy to implement correctly and be resistant against cache and side-channel attacks;
- *Error sampling* – although an ideal model of error distribution is Gaussian, with an appropriate standard deviation, FrodoKEM uses a sampled distribution computed via a small lookup table and a few random bits, thus yet amplifying the security against mentioned cache and side-channel attacks;
- *Key encapsulation without reconciliation* – functions that prepare secret key to distribution are based on simple Regev’s encryption scheme, which considers transmitting secret bits by simply encoding them with pseudorandom values that the receiver can approximately subtract away. Simplicity comes together with more simplicity, because these operations do not require later bandwidth-saving optimizations which take place in sister algorithms that make use of structure lattices (i.e. NewHope scheme – RLWE based);
- *Flexibility for parameters* – as the plain LWE does not impose many restrictions on its parameters, it is possible to suit them into any needed implementation, as far as thus generated keys meet security criteria. For instance, using a larger LWE modulus and appropriate dimensions of matrices, FrodoPKE (the general scheme rather than specialized version – FrodoKEM) can easily support a large number of homomorphic additions, or multiplications by (small) public scalars;
- *Dynamically generated public matrices* – in order to avoid any possibility of backdoors or any kinds of attacks against algorithm structure, a fresh matrix A is generated for every key generation. This slows down the encryption process but brings more security versus having a public matrix with fixed values (which would on the other hand be an acceleration for entire process).

With these top-level assumptions and overview, let us delve straight into step-by-step explanation of these algorithms using combination of mathematical language and pseudo-code. The FrodoKEM documentation (Erdem Alkim, 2019) starts with general FrodoPKE structure, which satisfies IND-CPA level of security. FrodoKEM structure derives from FrodoPKE public encryption scheme by applying the Fujisaki-Okamoto transform with implicit rejection (FO^\perp), which constructs an IND-CCA-secure KEM from IND-CPA public key encryption scheme with three different hash functions (Eiichi Fujisaki, 1999). Thus, FrodoKEM functions have more complex structure and more parameters, but the base remains the same. Definitions of these specific bottom-level parameters of FrodoKEM are presented below.

- $q = 2^D$ – integer modulus with exponent $D \leq 16$;

- n, \bar{m}, \bar{n} – integer matrix dimensions with $n \equiv 0 \pmod{8}$;
- $B \leq D$ – number of bits encoded in each matrix entry;
- $l = B * \bar{m} * \bar{n}$ – the length of bit strings to be encoded in an $\bar{m} \times \bar{n}$ matrices;
- $len_\mu = l$ – the bit length of messages;
- $M = \{0,1\}^{len_\mu}$ – the message space;
- len_{seedA} – the bit length of seeds used for pseudorandom matrix generation;
- len_{seedSE} – the bit length of seeds used for pseudorandom bit generation for error sampling;
- Gen – pseudorandom matrix generation algorithm; explained later in this section;
- T_χ – distribution table for sampling;
- len_s – the length of the bit vector s used for pseudorandom shared secret generation in the event of decapsulation failure in the FO^\perp transform;
- len_z – the bit length of seeds used for pseudorandom generation of $seed_A$;
- len_k – the bit length of intermediate shared secret k in the FO^\perp transform;
- len_{pkh} – the bit length of the hash $G_I(pk)$ of the public key in the FO^\perp transform;
- len_{ss} – the bit length of shared secret ss in the FO^\perp transform;

The Table 5. FrodoKEM parameters' specific values presents specific values for these parameters as used in Microsoft's implementation (Microsoft, 2019).

	Frodo-640	Frodo-976	Frodo-1344
D	15	16	16
q	32768	65536	65536
n	640	976	1344
$\bar{n} = \bar{m}$	8	8	8
B	2	3	4
len_{seedA}	128	128	128
len_z	128	128	128
$len_\mu = l$	128	192	256
len_{seedSE}	128	192	256
len_s	128	192	256
len_k	128	192	256
len_{pkh}	128	192	256
len_{ss}	128	192	256
len_χ	16	16	16

Table 5. FrodoKEM parameters' specific values (Erdem Alkim, 2019)

With these parameters, now a step-by-step explanation of key generation, encapsulation and decapsulation algorithms will be presented (Erdem Alkim, 2019). Note that these functions have certain subroutines for specific operations, such as Unpack() or Encode(). They will be explored later in this section.

Algorithm 1. FrodoKEM Key generation

- FrodoKEM.KeyGeneration()
 - Input: None
 - Output: Key pair (pk, sk') with $pk \in \{0,1\}^{len_{seedA} + D * n * \bar{n}} \times Z_q^{n \times \bar{n}} \times \{0,1\}^{len_{pkh}}$
 - 1) Choose uniformly random seeds $s || seed_{SE} || z <- U(\{0,1\}^{len_s + len_{seedSE} + len_z})$
 - 2) Generate pseudorandom seed: $seed_A <- \text{SHAKE}(z, len_{seedA})$
 - 3) Generate the matrix $A \in Z_q^{n \times \bar{n}}$ via $A <- \text{Frodo.Gen}(seed_A)$

- 4) Generate pseudorandom bit string $(r^{(0)}, r^{(1)}, \dots, r^{(2n\bar{n}-1)}) \leftarrow \text{SHAKE}(0x5F||\text{seed}_{SE}, 2m\bar{n} \times \text{len}_\chi)$
- 5) Sample error matrix $S \leftarrow \text{Frodo.SampleMatrix}((r^{(0)}, r^{(1)}, \dots, r^{(2n\bar{n}-1)}), n, \bar{n}, T\chi)$
- 6) Sample error matrix $E \leftarrow \text{Frodo.SampleMatrix}((r^{(n\bar{n})}, r^{(n\bar{n}+1)}, \dots, r^{(2n\bar{n}-1)}), n, \bar{n}, T\chi)$
- 7) Compute $B \leftarrow AS + E$
- 8) Compute $b \leftarrow \text{Frodo.Pack}(B)$
- 9) Compute $\text{pkh} \leftarrow \text{SHAKE}(\text{seed}_A || b, \text{len}_{\text{pkh}})$
- 10) *return*: public key $\text{pk} \leftarrow \text{seed}_A || b$ and secret key $\text{sk}' \leftarrow (s || \text{seed}_A || b, S, \text{pkh})$

Algorithm 2. FrodoKEM key encapsulation

- FrodoKEM.Encapsulation()
 - *Input*: Public key $\text{pk} = \text{seed}_A || b \in \{0,1\}^{\text{len}_{\text{seed}_A} + D \cdot n \cdot \bar{n}}$
 - *Output*: Ciphertext $c_1 || c_2 \in \{0,1\}^{(\bar{m} \cdot \bar{n} + \bar{m} \cdot \bar{n})D}$ and shared secret $\text{ss} \in \{0,1\}^{\text{len}_{\text{ss}}}$
 - 1) Choose a uniformly random key $\mu \leftarrow U(\{0,1\}^{\text{len}_\mu})$
 - 2) Compute $\text{pkh} \leftarrow \text{SHAKE}(\text{pk}, \text{len}_{\text{pkh}})$
 - 3) Generate pseudorandom values $\text{seed}_{SE} || k \leftarrow \text{SHAKE}(\text{pkh} || \mu, \text{len}_{\text{seed}_{SE}} + \text{len}_k)$
 - 4) Generate pseudorandom bit string $(r^{(0)}, r^{(1)}, \dots, r^{(2\bar{m}n + \bar{m}\bar{n} - 1)}) \leftarrow \text{SHAKE}(0x96 || \text{seed}_{SE}, 2\bar{m}n + \bar{m}\bar{n} \times \text{len}_\chi)$
 - 5) Sample error matrix $S \leftarrow \text{Frodo.SampleMatrix}((r^{(0)}, r^{(1)}, \dots, r^{(\bar{m}n - 1)}), \bar{m}, n, T\chi)$
 - 6) Sample error matrix $E \leftarrow \text{Frodo.SampleMatrix}((r^{(\bar{m}n)}, r^{(\bar{m}n + 1)}, \dots, r^{(2\bar{m}n - 1)}), \bar{m}, n, T\chi)$
 - 7) Generate $A \leftarrow \text{Frodo.Gen}(\text{seed}_A)$
 - 8) Compute $B' \leftarrow S'A + E'$
 - 9) Compute $c_1 \leftarrow \text{Frodo.Pack}(B')$
 - 10) Sample error matrix $E'' \leftarrow \text{Frodo.SampleMatrix}((r^{(2\bar{m}n)}, r^{(2\bar{m}n + 1)}, \dots, r^{(2\bar{m}n + \bar{m}\bar{n} - 1)}), \bar{m}, \bar{n}, T\chi)$
 - 11) Compute $B \leftarrow \text{Frodo.Unpack}(b, n, \bar{n})$
 - 12) Compute $V \leftarrow S'B + E''$
 - 13) Compute $C \leftarrow V + \text{Frodo.Encode}(\mu)$
 - 14) Compute $c_2 \leftarrow \text{Frodo.Pack}(C)$
 - 15) Compute $\text{ss} \leftarrow \text{SHAKE}(c_1 || c_2 || k, \text{len}_{\text{ss}})$
 - 16) *return*: ciphertext $c_1 || c_2$ and shared secret ss

Algorithm 3. FrodoKEM Key decapsulation

- FrodoKEM.Decapsulation()
 - *Input*: Ciphertext $c_1 || c_2 \in \{0,1\}^{(\bar{m} \cdot n + \bar{m} \cdot \bar{n})D}$, secret key $\text{sk}' = (s || \text{seed}_A || b, S, \text{pkh}) \in \{0,1\}^{\text{len}_s + \text{len}_{\text{seed}_A} + D \cdot n \cdot \bar{n}} \times \mathbb{Z}_q^{n \times \bar{n}} \times \{0,1\}^{\text{len}_{\text{pkh}}}$
 - *Output*: Shared secret $\text{ss} \in \{0,1\}^{\text{len}_{\text{ss}}}$
 - 1) $B' \leftarrow \text{Frodo.Unpack}(c_1)$

- 2) $C \leftarrow \text{Frodo.Unpack}(c_2)$
- 3) Compute $M \leftarrow C - B'S$
- 4) Compute $\mu' \leftarrow \text{Frodo.Decode}(M)$
- 5) Parse $pk \leftarrow \text{seed}_A || b$
- 6) Generate pseudorandom values $\text{seed}_{SE}' || k' \leftarrow \text{SHAKE}(pkh || \mu, \text{len}_{\text{seed}_{SE}} + \text{len}_k)$
- 7) Generate pseudorandom bit string $(r^{(0)}, r^{(1)}, \dots, r^{(2\bar{m}n + \bar{m}\bar{n} - 1)}) \leftarrow \text{SHAKE}(0x96 || \text{seed}_{SE}', 2\bar{m}n + \bar{m}\bar{n} \times \text{len}_\chi)$
- 8) Sample error matrix $S' \leftarrow \text{Frodo.SampleMatrix}((r^{(0)}, r^{(1)}, \dots, r^{(\bar{m}n - 1)}), \bar{m}, n, T\chi)$
- 9) Sample error matrix $E' \leftarrow \text{Frodo.SampleMatrix}((r^{(\bar{m}n)}, r^{(\bar{m}n + 1)}, \dots, r^{(2\bar{m}n - 1)}), \bar{m}, n, T\chi)$
- 10) Generate $A \leftarrow \text{Frodo.Gen}(\text{seed}_A)$
- 11) Compute $B'' \leftarrow S'A + E'$
- 12) Sample error matrix $E'' \leftarrow \text{Frodo.SampleMatrix}((r^{(2\bar{m}n)}, r^{(2\bar{m}n + 1)}, \dots, r^{(2\bar{m}n + \bar{m}\bar{n} - 1)}), \bar{m}, \bar{n}, T\chi)$
- 13) Compute $B \leftarrow \text{Frodo.Unpack}(b, n, \bar{n})$
- 14) Compute $V \leftarrow S'B + E''$
- 15) Compute $C' \leftarrow V + \text{Frodo.Encode}(\mu')$
- 16) If $B' || C = B'' || C'$ then
 - a. return: shared secret $ss \leftarrow \text{SHAKE}(c_1 || c_2 || k', \text{len}_{ss})$
 - else:
 - b. return: shared secret $ss \leftarrow \text{SHAKE}(c_1 || c_2 || ss, \text{len}_{ss})$

These algorithms all contain several steps consisting i.e. generation of pseudorandom strings, filling different matrices with that data and operating on them. Encapsulation and Decapsulation algorithms are also symmetric, they even use the same variables for storing results. Note that amongst cryptographic primitives which FrodoKEM uses there is a SHAKE function, which points to specifically either SHAKE128/SHAKE256 (depending on chosen level of security) – or AES128 function. Either of these primitives can be used for certain operations in FrodoKEM, such as instantiation pseudorandom matrices A , for instance. The SHAKE and underlying Keccak algorithm use specific padding in order to pad out input strings to a multiple of the rate. SHAKE depends on Keccak's padding for domain separation when passed inputs of different lengths, otherwise it prepends some distinct bytes for this purpose. These separators (0x5F, 0x96) have bit patterns that aim to harden algorithm against bit flipping attacks turning one separator into the other. As to the subroutines, the matrix encoding of bit strings will be firstly discussed. $\text{Frodo.Encode}()$ takes a bit string of length $l = B * \bar{m} * \bar{n}$ as $\bar{m} \times \bar{n}$ matrices with entries in Z_q and encodes it to B -bit sub-strings sequentially and filling the matrix row by row and entry-wise using $\text{ec}()$ function. Opposite function, $\text{Frodo.Decode}()$ extracts B bits from each entry, thus decoding $\bar{m} \times \bar{n}$ matrix K into a bit string of length l by applying $\text{dc}()$. Below are algorithms for these functions:

Algorithm 4.5. FrodoKEM Encode and Decode

$\text{Frodo.Encode}()$

Input: Bit string $k \in \{0,1\}^l, l = B * \bar{m} * \bar{n}$

Output: Matrix $K \in Z_q^{\bar{m} \times \bar{n}}$

- 1) for ($i=0; i < \bar{m}; i \leftarrow i+1$) do
 - a. for ($j=0; j < \bar{n}; j \leftarrow j+1$) do
 - i. $k \leftarrow \sum_{h=0}^{B-1} k_{(i*\bar{n}+j)B+h} * 2^h$

$\text{Frodo.Decode}()$

Input: Matrix $K \in Z_q^{\bar{m} \times \bar{n}}$

Output: Bit string $k \in \{0,1\}^l, l = B * \bar{m} * \bar{n}$

- 1) for ($i=0; i < \bar{m}; i \leftarrow i+1$) do
 - a. for ($j=0; j < \bar{n}; j \leftarrow j+1$) do
 - i. $k \leftarrow \text{dc}(K_{i,j}) = \lfloor K_{i,j} * 2^B / q \rfloor \bmod 2^B$

ii. $K_{i,j} \leftarrow \text{ec}(k) = k * q / 2^B$
 2) return $K = (K_{i,j})_{0 \leq i < \bar{m}, 0 \leq j < \bar{n}}$

ii. $k = \sum_{h=0}^{B-1} k_h * 2^h$ where $k_h \in \{0,1\}$

iii. for $(h=0; h < B; h \leftarrow h+1)$ do
 $k_{(i*\bar{n}+j)B+l} \leftarrow k_h$

2) return k

Next to discuss are packing and unpacking algorithms that transform matrices with entries in Z_q to bit strings, and vice versa. Frodo.Pack() does the latter by concatenating D-bit matrix coefficients. In the software implementation, resulting bit string is stored as a byte array, padding with zero bits to make the length a multiple of 8.

Algorithm 5. FrodoKEM Pack and Unpack

Frodo.Pack()

Input: Matrix $C \in Z_q^{n_1 \times n_2}$

Output: Bit string $b \in \{0,1\}^{D*n_1*n_2}$

1) for $(i=0; i < n_1; i \leftarrow i+1)$ do
 a. for $(j=0; j < n_2; j \leftarrow j+1)$ do
 i. $C_{i,j} = \sum_{h=0}^{D-1} C_h * 2^h$ where $C_h \in \{0,1\}$
 ii. for $(h=0; h < D; h \leftarrow h+1)$ do
 $b_{(i*n_2)D+h} \leftarrow C_{D-1-h}$
 2) return b

Frodo.Unpack()

Input: Bit string $b \in \{0,1\}^{D*n_1*n_2}$, n_1 , n_2

Output: Matrix $C \in Z_q^{n_1 \times n_2}$

1) for $(i=0; i < n_1; i \leftarrow i+1)$ do
 a. for $(j=0; j < n_2; j \leftarrow j+1)$ do
 i. $C_{i,j} \leftarrow \sum_{h=0}^{D-1} b_{(i*n_2+j)D+h} * 2^{D-1-h}$
 2) return C

The error distribution χ used in FrodoKEM is a discrete, symmetric distribution on Z (approximation of a rounded continuous Gaussian distribution). The support of χ is $S_\chi = \{-s, -s+1, \dots, -1, 0, 1, \dots, s-1, s\}$ for a positive integer s . Sampling from this distribution is done via function Frodo.Sample(). Given a string of len_χ uniformly random bits $r \in \{0,1\}^{\text{len}_\chi}$ and a distribution table T_χ , this function returns a sample e from the distribution χ . It is important to perform this sampling in constant time to avoid exposing timing side-channels. This is why the algorithm does a complete loop through the entire table T_χ . Discrete probability density function used in sampling is described by a table $T_\chi = (T_\chi(0), T_\chi(1), \dots, T_\chi(s))$. For the purposes of FrodoKEM, a $n_1 \times n_2$ matrix of $n_1 n_2$ samples from the error distribution is sampled on output of a $(n_1 n_2 * \text{len}_\chi)$ -bit string, written as a sequence of $n_1 n_2$ bit vectors of len_χ each, by using Frodo.Sample() on each corresponding len_χ -bit substring and a distribution table T_χ to sample the matrix entry $E_{i,j}$ inside Frodo.SampleMatrix() function. Structure of these functions is as follows:

Algorithm 6. FrodoKEM Sample and SampleMatrix

Frodo.Sample()

Input: A random bit string $r \in \{0,1\}^{\text{len}_\chi}$, T_χ

Output: A sample $e \in Z$

1) $t \leftarrow \sum_{i=1}^{\text{len}_\chi-1} r_i * 2^{i-1}$
 2) $e \leftarrow 0$
 3) for $(z=0; z < s; z \leftarrow z+1)$ do
 a) if $t > T_\chi(z)$ then
 $e \leftarrow e+1$

Frodo.SampleMatrix()

Input: A random bit string $r \in \{0,1\}^{n_1 n_2 \text{len}_\chi}$, T_χ

Output: A sample $E \in Z^{n_1 \times n_2}$

1) for $(i=0; i < n_1; i \leftarrow i+1)$ do
 a) for $(j=0; j < n_2; j \leftarrow j+1)$ do
 i. $E_{i,j} = \text{Frodo.Sample}(r^{(i*n_2+j)}, T_\chi)$
 2) return E

- 4) $e \leftarrow (-1)^{r_0} * e$
- 5) return e

The last puzzle for FrodoKEM is function that creates a pseudorandom matrix $A \in Z_q^{m \times n}$, given $seed_A \in \{0,1\}^{len_{seedA}}$ and a dimension $n \in Z$ as a parameter. Function is named Frodo.Gen() and has two options – AES128 (default in later discussed Microsoft’s implementation), and SHAKE128. Algorithms for both variants are listed below:

Algorithm 7. FrodoKEM Gen using AES128 or SHAKE128

Frodo.Gen() using AES128

Input: Seed $seed_A \in \{0,1\}^{len_{seedA}}$

Output: Matrix $A \in Z_q^{m \times n}$

- 1) for ($i=0; i < n; i \leftarrow i+1$) do
 - a) for ($j=0; j < n; j \leftarrow j+8$) do
 - i. $b \leftarrow \langle i \rangle \| \langle j \rangle \| 0 \dots 0 \in \{0,1\}^{128}$ where $\langle i \rangle, \langle j \rangle \in \{0,1\}^{16}$
 - ii. $\langle c_{i,j} \rangle \| \langle c_{i,j+1} \rangle \| \dots \| \langle c_{i,j+7} \rangle \leftarrow \text{AES128}_{seed_A}(b)$ where each $\langle c_{i,k} \rangle \in \{0,1\}^{16}$
 - iii. for ($k=0; k < 8; k \leftarrow k+1$) do
 $A_{i,j+k} \leftarrow c_{i,j+k} \bmod q$
- 2) return A

Frodo.Gen() using SHAKE128

Input: Seed $seed_A \in \{0,1\}^{len_{seedA}}$

Output: Matrix $A \in Z_q^{m \times n}$

- 1) for ($i=0; i < n; i \leftarrow i+1$) do
 - a) $b \leftarrow \langle i \rangle \| seed_A \in \{0,1\}^{16+len_{seedA}}$ where $\langle i \rangle \in \{0,1\}^{16}$
 - b) $\langle c_{i,0} \rangle \| \langle c_{i,1} \rangle \| \dots \| \langle c_{i,n-1} \rangle \leftarrow \text{SHAKE128}(b, 16n)$ where each $\langle c_{i,j} \rangle \in \{0,1\}^{16}$
 - c) for ($j=0; j < n; j \leftarrow j+1$) do
 $A_{i,j} \leftarrow c_{i,j} \bmod q$
- 2) return A

2.4.3. Motivation for security

FrodoKEM achieves security on several different levels. The IND-CCA security is achieved through transformation of the IND-CPA-secure FrodoPKE. IND-CCA often refers to actually IND-CCA2 – the abbreviation that comes from IND-distinguishability under adaptive Chosen Ciphertext Attack. As puzzling as it sounds, it simply refers to giving adversary further knowledge of secure communication by being able to question hypothetical “oracle” as to the validity of chosen ciphertexts. In IND-CCA1 situation adversary does not have this advantage but is still able to encrypt or decrypt arbitrary messages before obtaining the challenge ciphertext. The first level – IND-CCA – states that adversary does not have any advantages at all and must decide i.e. which of two messages was encrypted using chosen cryptosystem (Rogaway, 1999). Additionally, another notion of security used through literature is non-malleability (NM-) under chosen attack model (-CPA, -CCA, -CCA2). It practically means a situation where an adversary cannot selectively modify ciphertexts but is only able to attack using DoS (Denial of Service) or replace entire message.

To achieve certain levels of security, top-level FrodoKEM parameters (which are: secret, public keys, ciphertext and shared secret) must have adequate size in bytes in order to meet NIST proposals’ criteria for quantum secure cryptosystems. These parameters are presented in the *Table 6*. FrodoKEM Key sizes in bytes. Note that NIST’s decapsulation API does not include an input for the public key, as it needs to be included as a part of the secret key.

Scheme	Secret key (sk)	Public key (pk)	Ciphertext (c)	Shared secret (ss)
--------	---------------------	---------------------	--------------------	------------------------

<i>FrodoKEM-640</i>	19 888 (10 256 + 9 616 + 16)	9 616 (16 + 9 600)	9 720 (9 600 + 120)	16
<i>FrodoKEM-976</i>	31 296 (15 640+15 632+16)	15 632 (16 + 15 616)	15 744 (15 616 + 128)	24
<i>FrodoKEM-1344</i>	43 088 (21 536+21 520 + 32)	21 520 (16 + 21 504)	21 632 (21 504 + 128)	32

Table 6. FrodoKEM Key sizes in bytes

Constraints for FrodoKEM's bottom-level parameters can also be chosen to guarantee as good error-free and secure encryption as possible or giving a bounded number of homomorphic operations. *Table 7.* FrodoKEM key parameters sizes, ranges and approximated security presents set of these parameters, together with estimated Classical (C) and Quantum (Q) security in bits (Erdem Alkim, 2019):

	n	q	Σ	Support of χ	B	$\bar{m} \times \bar{n}$	Failure probability	c size (bytes)	Security C Q	
<i>Frodo-640</i>	640	2^{15}	2.8	[-12...12]	2	8×8	$2^{-138.7}$	9 736	144	103
<i>Frodo-976</i>	976	2^{16}	2.3	[-10...10]	3	8×8	$2^{-199.6}$	15 768	209	150
<i>Frodo-1344</i>	1344	2^{16}	1.4	[-6...6]	4	8×8	$2^{-252.5}$	21 664	274	196

Table 7. FrodoKEM key parameters sizes, ranges and approximated security

Although failure probability is extremely small, it is still worth remembering that FrodoKEM is not entirely deterministic, due to its lattice-related structure. Nevertheless, it achieves strong levels of security both against classical and quantum cryptanalysis, and under certain attack scenarios, such as CPA or CCA. All this makes FrodoKEM a very strong candidate for quantum secure cryptography. One has to bear in mind enormous key sizes of this cryptosystem, which might be not suitable for mobile architectures. However, Microsoft has already provided implementation of FrodoKEM for ARM processors, which, even with mentioned sizes, is not very far from perfect.

2.5. Supersingular Isogeny Diffie-Hellman (SIDH)

Supersingular Isogeny Diffie Hellman (SIDH) key exchange protocol was proposed by Jao and De Feo in 2011. This makes it one of the newest quantum-secure protocols, so its security has to be yet thoroughly tested. Costello has provided a following account on elliptic-curve cryptography (Craig Costello):

“The field of elliptic-curve cryptography is somewhat notorious for using quite a bit of arcane math. Despite its freshness its security properties are promising, since best known classical and quantum attacks against the underlying problem are both exponential in size of the underlying finite field.” (Craig Costello)

And later added (Craig Costello):

“Complexity of computing isogenies between supersingular elliptic curves (in 2010 there was created a quantum algorithm that computes isogenies between ordinary (i.e. non-singular) ECs in sub-exponential time) makes current SIDH key sizes significantly smaller than those used in FrodoKEM and other post-quantum key exchange candidates. “ (Craig Costello)

For the SIDH scheme, secret keys are a chain of isogenies and public keys are curves. When Alice and Bob combine this information, they acquire curves that are different, but have the same j -invariant. This term

will be explained later. To summarize, *Table 8*. Comparison of DH variations presents main differences between traditional DH algorithm and its two most common EC-based variations. Most important is that the key exchange scheme remained the same, even though under-the-hood operations evolved significantly.

	DH	ECDH	SIDH
Elements	Integers g modulo prime	Points P in curve group	Curves E in isogeny class
Secrets	exponents x	scalars k	isogenies φ
Computations	$g, x \Rightarrow g^x$	$k, P \Rightarrow [k]P$	$\varphi, E \Rightarrow \varphi(E)$
Hard Problem	given g, g^x find x	given $P, [k]P$ find k	given $E, \varphi(E)$ find φ

Table 8. Comparison of DH variations

This paragraph is focused on the analysis of SIDH algorithm and its underlying mathematical problem – constructing isogenies between supersingular ECs.

2.5.1. Morphisms

Morphism is a structure-preserving map from one mathematical structure to another of the same type. In set theory morphisms are functions, in linear algebra – linear transformations, in group theory they are called group homomorphisms. In case of elliptic curves, morphism is a function described by ratios of polynomials that maps points on E_1 to points on E_2 ($f: E_1 \Rightarrow E_2$) (Silverman, 2009).

In most cases objects are sets, whereas morphisms are functions that change object into another object. These objects, before and after applying function, are called *domain* and *codomain* respectively. Morphisms have a *partial binary operation* called composition. Composition of two morphisms f and g is defined precisely when the target of morphism f is the source of morphism g . It is denoted $g \circ f$. The source of composition is the source of f , and the target is the target of g . Composition satisfies two axioms (Hazewinkel, 2001):

- *Identity* – there is an identity morphism of object X , denoted $\text{id}_X: X \rightarrow X$, such that for every morphism $f: A \Rightarrow B$ there is $\text{id}_B \circ f = f = f \circ \text{id}_A$
- *Associativity* – when all compositions are defined side of objects in composition is of no difference $h \circ (g \circ f) = (h \circ g) \circ f$

There are in fact a few special cases of morphisms. Monomorphism is a function $f: X \rightarrow Y$ if $f \circ g_1 = f \circ g_2$ implies $g_1 = g_2$ for all morphisms $g_1, g_2: Z \rightarrow X$. Isomorphism $f: X \rightarrow Y$ in turn, must have another morphism $g: Y \rightarrow X$ such that $f \circ g = \text{id}_Y$ and $g \circ f = \text{id}_X$. Isomorphism of EC is a morphism that satisfies $f(0_E) = 0_E$ and whose inverse (over the algebraic structure) is also a morphism. In general, there is an elliptic curve E' called the *quadratic twist* of E , if it satisfies $\#E'(F_q) = q + 1 + t$ ($\#$ symbol denotes *order* of elliptic curve – more on this in isogenies section below) and it is isomorphic to E (where isomorphism is defined over quadratic extension F_{q^2}).

Another important term in case of elliptic curves is j -invariant of an elliptic curve. It is a function of a *complex variable* τ , or a modular function of weight zero for $SL(2, \mathbb{Z})$ defined on the upper half-plane of complex numbers. J -invariant of an elliptic curve is defined as (Steven D. Galbraith):

$$j(E) = 1728 \frac{4A^3}{4A^3 + 27B^2} \quad 2.20$$

Where variables A and B are coefficients taken from EC Weierstrass' equation. Finally, all supersingular ECs over the algebraic closure of F_p are isomorphic to some supersingular curves over F_{p^2} .

2.5.2. Isogenies

An isogeny ϕ is a function transforming one elliptic curve into another, or just a homomorphism with additional structure dealing with geometry of each curve in this relation. Isogeny $\phi: E_1 \rightarrow E_2$ is in other words a morphism (rational map), or a non-constant rational, between algebraic varieties which is given locally by polynomials and is preserving the identity i.e. $\phi(\infty_1) = \infty_2$ (blperez1, 2018). A regular map whose inverse is also regular is called *biregular*, and these are isomorphisms in the category of algebraic varieties. The image curve under the isogeny is called the quotient curve. Quotients here are generally associated to a surjective quotient map (Khan Academy) – and the map here is called an isogeny. Main case of an isogeny is the multiplication by n map $[n]: E \rightarrow E$. In Elliptic Curve Discrete Logarithm (ECDL) the problem is finding n when given points P and $Q = [n]P$. The problem in SIDH is determining an isogeny when given points are P and $Q = \phi(P)$. Shor's algorithm will solve this problem in a polynomial time (Steven D. Galbraith).

The order of a group in group theory is its cardinality i.e. the number of elements in its set. Similarly, the degree/order of an (separate) isogeny is its degree as a rational map, or a number of elements in its kernel. Its denoted with symbol $\#$ (i.e. $\#E(F_q) = q + 1 - t$). Every size of degree greater than 1 can be factored into a composition of isogenies of prime degree over F_q . An endomorphism of an elliptic curve E_1 defined over F_q is an isogeny $E_1 \rightarrow E$ defined over F_{q^m} for some m . The set of endomorphisms of E together with the zero map forms a ring under the operations of pointwise addition and composition. This ring is called the endomorphism ring of E_1 (denoted as $\text{end}(E_1)$). Problem of computing $\text{end}(E')$ and the problem of computing an isogeny $\phi: E \rightarrow E'$ are broadly equivalent. Explicit isogenies $\phi: E_0 \rightarrow E_0$ as rational functions is one way to represent $\text{end}(E')$ (David Jao/Luca De Feo).

Elliptic curve is called *supersingular* if p satisfies $p \mid t$, where t is an integer satisfying $|t| \leq 2\sqrt{q}$. For an elliptic curve over F_q , p is associated with q with the following equation: $q = p^a$. If p does not satisfy this condition, elliptic curve is called *ordinary*. Other conditions for supersingularity are associated with calculating the degree of elliptic curve $\#E(F_q) = q + 1 - t$:

- E is *supersingular* if $\#E(F_q) \equiv 1 \pmod{p}$
- For $n \in \mathbb{N}$ define $E[n] = \{P \in E(F_q) : [n]P = 0\}$.
 - $p \mid n$ then $\#E[n] = n^2$
 - If E is *supersingular* then $E[p] = \{0\}$
 - If E is *ordinary*, then $\#E[p] = p$

SIDH isogeny problem is:

Given $(E, R_2, S_2, E_A, R'_2, S'_2)$ to determine an isogeny $\phi_A: E \rightarrow E_A$ of degree $l_1 e_1$ such that $R'_2 = \phi_A(R_2)$ and $S'_2 = \phi_A(S_2)$, where let $R'_2 = \phi_A(R_2), S'_2 = \phi_A(S_2)$

The decisional version of the problem requires the following computations:

- Reduce the problem to the case of EC whose endomorphism ring is maximal. Having E_1 and E_2 , both of which have $q + 1 - t$ elements. Algorithm constructs a chain of isogenies from E_i to E'_i where $\text{end}(E'_i) = Q_K$ is the maximal order of $K = \mathbb{Q}(\sqrt{t^2 - 4q})$
- Try to construct an isogeny between E'_1 and E'_2

The general isogeny problem – given j and $j' \in F_q$ find an isogeny $\varphi: E \rightarrow E'$ (if possible – $j(E) = j$ and $j(E') = j'$). Decisional question of whether an isogeny exists or not is solvable in polynomial time: an isogeny $\varphi: E \rightarrow E'$ exists if and only if $\#E(F_q) = \#E'(F_q)$. Computing the points could be done in the same time. If one isogeny exists, however, then there are an infinite number of isogenies for these EC. Asking for specific isogeny, having i.e. minimal degree is much more difficult. Now considering the graph created by examining all the isogenies of this form from our starting curve, then all the isogenies from those curves, and so on. This graph turns out to be highly structured in the sense that if taking a random walk starting at first curve, the probability of hitting a specific other curve is negligibly small. In other words, the graph generated by examining all this isogenies is an expander graph. This property of expansion is precisely what makes isogeny-based cryptography secure.

2.5.2.1. Problems related to finding isogenies

In SIDH there are used isogeny classes with smooth orders. It makes computing rational isogenies of exponentially large (but smooth) degree effective as a process of composition of low degree isogenies (using Velu's formulas (Costello, 2010)). For instance, take two small prime numbers l_A and l_B , and integer cofactor f , and p being a prime of the form $p = l_A^{e_A} l_B^{e_B} f \pm 1$. Now a supersingular elliptic curve defined over F_{q^2} of order $[l_A^{e_A} l_B^{e_B} f \pm 1]$ can be constructed. For $l \in \{l_A, l_B\}$ and $e \in \{e_A, e_B\}$ - the corresponding exponent, full l^e -torsion group on E is defined over F_{p^2} i.e. $E[l^e] \subseteq E(F_{p^2})$. Since l is coprime to p , $E[l^e] \cong (Z/l^e Z) \times (Z/l^e Z)$. Let $P, Q \in E[l^e]$ be two points that generate $E[l^e]$ such that the above isomorphism is given by $(Z/l^e Z) \times (Z/l^e Z) \Rightarrow E[l^e]$, $(m, n) \Rightarrow [m]P + [n]Q$ (Hazewinkel, 2001). In general, SIDH secret keys are degree l^e isogenies of the base curve E and are a bijection with cyclic subgroups of order l^e that form their kernels. A point $[m]P + [n]Q$ has full order l^e if and only if at least either m or n are not divisible by l . Since distinct cyclic subgroups only intersect in points of order less than l^e and all full-order points in a single subgroup are coprime multiplies of one such point, there are $l^{e-1} (l+1)$ distinct cyclic subgroups of order l^e (Craig Costello).

2.5.3. Motivation for security

Microsoft has developed an implementation of SIDH ephemeral SIDH which provides 128 bits of quantum security and 192 bits of classical security. It uses 48-byte private keys to provide 564-byte ephemeral DH public keys.

Microsoft's implementation is designed to run in *constant-time* to resist various algorithm-structure attacks – such as timing and cache timing attacks (Steven D. Galbraith).

Isogeny-based cryptography has extremely small key sizes compared to other post-quantum schemes, using only 330 bytes for public keys. Unfortunately, SIDH takes a lot more time to compute shared secret than other post-quantum candidates. On the other hand, SIDH supports perfect forward secrecy, which is not something other post-quantum cryptosystems possess.

2.5.4. Algorithm structure

The public parameters in SIDH are the supersingular curve E_0/F_{p^2} whose group order is $(l_A^e l_B^e f)^2$, two independent points P_A and Q_A that generate $E_0[l_A^e]$, and two independent points P_B and Q_B that generate $E_0[l_B^e]$. Keys are generated using Montgomery ladder algorithm and the Okeya-Sakurai strategy.

Computing public key is starts with the choice of two secret integers $m_A, n_A \in \mathbb{Z}/l_A^e \mathbb{Z}$ which are both not divisible by l_A , such that $R_A = [m_A]P_A + [n_A]Q_A$ has order l_A^e .

Computing secret key is done by computing degree l_A^e isogeny $\varphi_A: E_0 \Rightarrow E_A$ whose kernel is R_A and public key is the isogenous curve E_A together with the image points $\varphi_A(P_B)$ and $\varphi_A(Q_B)$.

The other party does calculate its keypair in a similar way:

Choose two secret integers $m_B, n_B \in \mathbb{Z}/l_B^e \mathbb{Z}$, both not divisible by l_B such that $R_B = [m_B]P_B + [n_B]Q_B$ has order l_B^e . Secret key is computed as the degree l_B^e isogeny $\varphi_B: E_0 \Rightarrow E_B$ whose kernel is R_B and public key is E_B together with $\varphi_B(P_A)$ and $\varphi_B(Q_A)$.

Computing the shared secret. Party A uses its secret integers and party B's public key to compute the degree l_A^e isogeny $\varphi'_B: E_B \Rightarrow E_{AB}$ whose kernel is the point $[m_B] \varphi_B(P_B) + [n_B] \varphi_B(Q_B) = \varphi_A(R_B)$. E_{BA} and E_{AB} are isomorphic, so party A and B can compute a shared secret as the common j-invariant $j(E_{BA}) = j(E_{AB})$ (Huseyin Hisil).

In short, the set-up for SIDH is as follows (Craig Costello):

Algorithm 8. SIDH Key generation

1. Find distinct, small primes l_1, l_2 and choose $e_1, e_2 \in \mathbb{N}$ such that $l_1^{e_1} \sim l_2^{e_2} \sim 2^\lambda$, where λ is some security parameter
2. Next choose a random integer $f \in \mathbb{N}$ until $p = l_1^{e_1} * l_2^{e_2} * f \pm 1$ is prime
3. Choose E to be supersingular EC over \mathbb{F}_{p^2} (Having an efficient algorithm to do this – Broker) such that $E(\mathbb{F}_{p^2})$ has group structure a product of two cyclic groups of order $l_1^{e_1} * l_2^{e_2} * f$
4. Fix points $R_1, S_1 \in E[l_1^{e_1}]$ such that the group $\langle R_1, S_1 \rangle$ generated by R_1 and S_1 is the whole group $E[l_1^{e_1}]$.
5. Similarly, choose $R_2, S_2 \in E[l_2^{e_2}]$
6. SIDH system params are: (E, R_1, S_1, R_2, S_2)

SIDH then computes shared secret as follows:

Algorithm 9. Computing shared secret in SIDH

1. Party A chooses a secret subgroup of $E[l_1^{e_1}]$ by choosing an integer $0 \leq a \leq l_1^{e_1}$ and setting $T_1 = R_1 + [a]S_1$
2. Party A computes an isogeny $\varphi_A: E \rightarrow E_A$ with kernel generated by T_1 and publishes $(E_A, \varphi_A(R_2), \varphi_A(S_2))$
3. Party B does same with $l_1^{e_1}$, producing $(E_B, \varphi_B(R_1), \varphi_B(S_1))$
4. To compute shared key, party A computes:
 - a. $T'_1 = \varphi_B(R_1) + [a] \varphi_B(S_1) = \varphi_B(R_1 + [a]S_1) = \varphi_B(T_1)$
 - b. And then computes an isogeny $\varphi'_A: E_B \rightarrow E_{AB}$ with kernel generated by T'_1

- c. The composition $\varphi'_A * \varphi_B : E \Rightarrow E_{AB}$ has kernel $\langle T_1, T_2 \rangle$
5. Party B computes $\varphi'_B : E_A \Rightarrow E'_{AB}$ with kernel $\langle \varphi_A(R_2) + [b] \varphi_A(S_2) \rangle$
6. Actual computed EC equations for A and B are not the same, but they are isomorphic so $j(E_{AB}) = j(E'_{AB})$. Hence, the shared key is $j(EAB)$

One example that will picture this algorithm is (Bernstein). In that paper, the author analyzed Curve25519, a specific elliptic curve function suitable for a variety of cryptographic applications. In general, Curve25519 user has a 32-byte secret and a 32-byte public key. That user will also calculate a 32-byte shared secret together with the party he is communicating with. The *Figure 8*. Example presentation of SIDH key exchange using Curve25519 presents the data flow from secret keys – through public keys – to a shared secret.

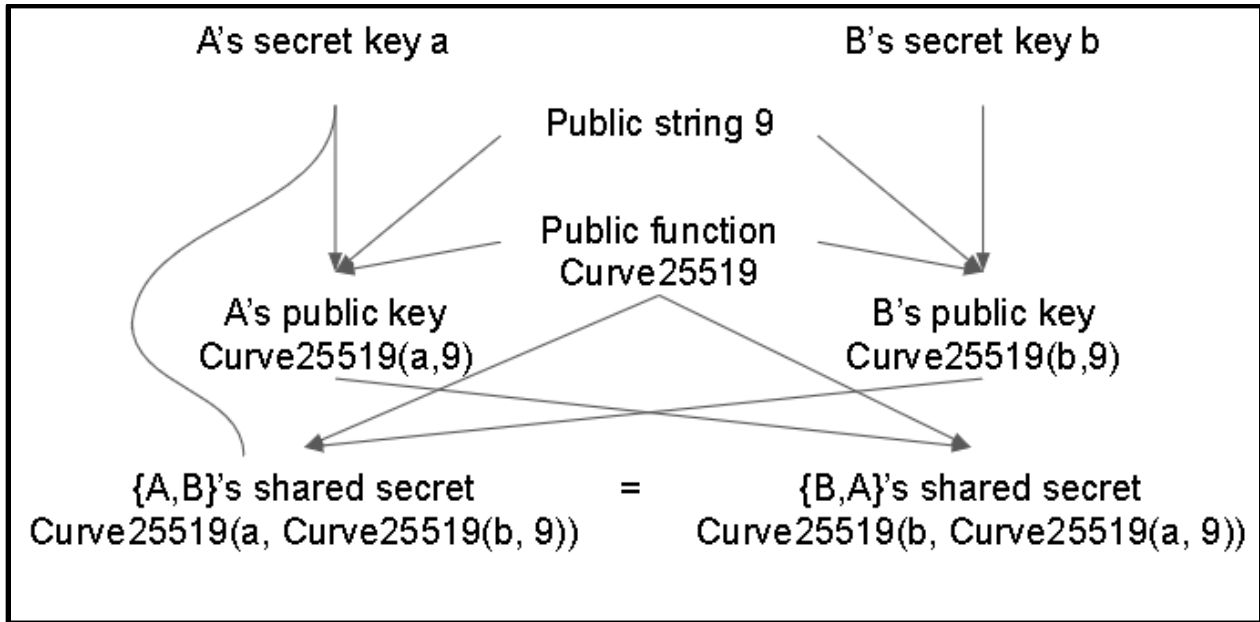


Figure 8. Example presentation of SIDH key exchange using Curve25519 (Bernstein)

2.5. Python 3.5.x

Python is an interpreted, dynamically typed programming language that was released in 1980s, it's successor – Python 2.0 in 2000 and current version (3.0) in 2008 which is not compatible with its predecessors. Day to day, latest version of Python is more and more widely used, and previous version (2.x) will no longer be supported after from 2020 (Wikipedia, 2019). Python is currently one of the most popular programming languages on the globe, steadily growing in popularity over the last years, according to TIOBE index (TIOBE, 2019). One of the reasons for this being the case is probably universality of Python – it supports multiple programming paradigms, i.e. procedural, OOP or functional programming, and another might be its extensibility with a comprehensive choice of external libraries. These reasons made Python very competitive amongst other languages and thus it can be spotted in actually every branch of IT industry – from AI to web development.

Python has several implementations. Most popular implementation, also referred to as “CPython” is written entirely in C programming language. It also means that most of external libraries for Python are written in C, most probably for efficiency reasons. Python itself is referred to as “not a speed daemon”, simple because most functions have hundreds of lines of C code inside, but this is the price for convenience and simplicity that offers Python – on a higher level of abstraction. Other popular implementations include Python in Java (Jython), in C# and so on.

Python has been chosen for implementations of algorithms analyzed in this project because of its versatility and the convenience it gives the programmer with specialized syntax and external libraries to carry matrix computations. Even if it will not be the fastest implementation, it would still bring educational value to the project and gives another perspective. As original FrodoKEM and SIDH are written in C, it is also an interesting comparison of low-level C functions with more abstract Python perspective.

3. Method section

This section focuses on implementation of mentioned key-exchange algorithms. It will briefly describe structure of own Python implementations, and their most important differences between original implementations written in other languages. Introduction of functionalities in GUI application created for the purpose of presentation of these algorithms is also a part of this section. Presentation application also includes simple communication service over the network, that will allow testing implemented key-exchange algorithms in a more practical environment.

3.5. Implementation of FrodoKEM in Python

Based on Microsoft’s work further discussed in **section 3.3.2**, a Python implementation of FrodoKEM algorithm has been created for the purpose of this dissertation. It is not only translation of low-level C functions into standardized, well defined, high-level functions that make the code more concise and understandable, but also extending existing paradigm into Object Oriented Programming (OOP) – thus giving the user convenience of interacting with ready objects that hide beneath all the necessary computations.

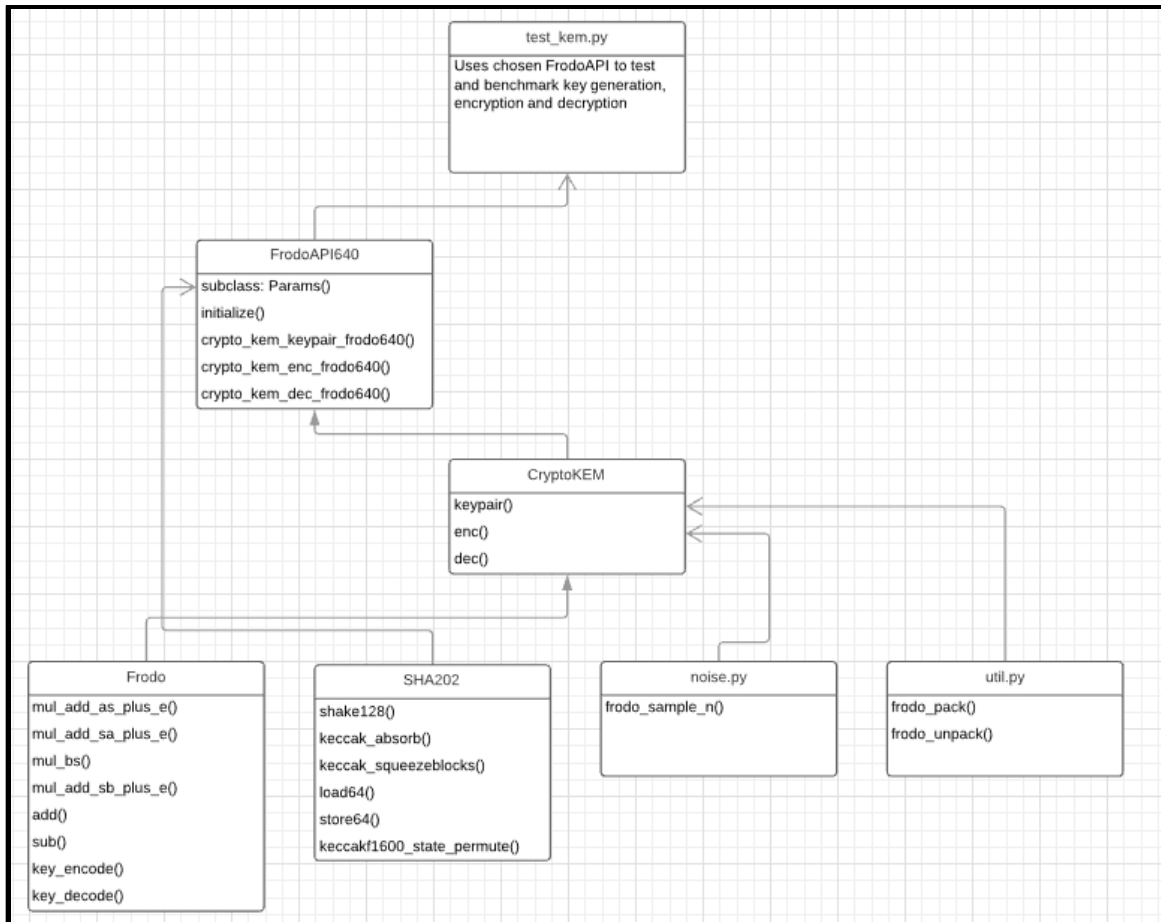


Figure 9. FrodoKEM Python implementation class structure

On the other side, main disadvantage of Python implementation is that it is times slower than its C equivalent – for the price of gained convenience. Figure 9. FrodoKEM Python implementation class structure presents structure of classes used in this FrodoKEM implementation, and a few files that contain complementary functions. Originally, there are three variations of the algorithm that target different level of security for NIST proposals (Alkim, 2019) – FrodoKEM-640, 976 and 1344. They differ mostly by size of generated keys, but also use slightly different functions for computations. For the purpose of this dissertation, only FrodoKEM-640 has been implemented with the default AES128 used to generate matrix A (according to Microsoft’s implementation – see section 2.4.2). However, adding other variations is simply a matter of copying its API to another file and changing parameters and name. All API classes inherit from CryptoKEM class – which provides implementation for calculating the pair of public and private keys, encapsulating them to send over the network and decapsulating after transport. It uses a few bottom-level classes for this purpose, amongst them Frodo – from which it inherits methods with abstract matrix arithmetic functions. Next class to mention is SHA202, which provides shake method for pseudo randomly generating matrix A that is used for key generation. This method is defined in FrodoAPI class, because depending on which variant is used it will used slightly different variation of shake function. Other files that implement complementary functions are noise.py for sampling and util.py for packing and unpacking vectors between different data formats.

3.5.1. Comparison with Microsoft's implementation

FrodoAPI640 contains algorithm parameters, such as length of keys and other relevant vectors enclosed in a subclass called Params. This class is later passed into all three functions, so that CryptoKEM can use these parameters. On object creation of FrodoAPI640 all the important vectors are created and zeroed – keys and encapsulated keys. Initialization is also repeated each time `crypto_kem_keypair_frodo640` function is called – so that before each key generation all vectors get cleared up.

Considering structure of the code itself, most important difference is a higher level of abstraction in Python implementation. Data types, pointers and loops are now handled by Python, which covers successfully all details, giving programmer a clean interface with vector objects and methods to operate on them. Moreover, Python is straightforwardly extendable with an extensive number of open-source libraries. Thus, most of the time programming a solution in Python is a matter of finding a proper library that has it already implemented. In case of scientific operations on matrices one of most popular choices is NumPy library. It also provides C-compatible data types for its arrays, which helped in keeping implementation closer to the original. *Figure 10*. Importing libraries – `frodo_macrify.py` presents an example of importing a few different libraries, and specific methods from each of imported libraries. This code sample is taken from `frodo_macrify.py` source file. Other libraries used generally in this project include Crypto which provided AES implementation and Secrets, that was used for generating *cryptographically secure* random tokens, according to its

```
from functools import reduce
from itertools import accumulate
from Crypto.Cipher import AES
from math import ceil
from numpy import zeros, uint16, frombuffer, uint8, uint64, array, sum, tile, \
    split, copyto, transpose, bitwise_and, array_split, hstack, repeat
```

Figure 10. Importing libraries – frodo_macrify.py

documentation (Python Library, n.d.). One advantage of using Python is simplicity of importing any libraries and using them in own project. It is possible to import only necessary functions from any library, and all dependencies are handled by Python interpreter. This way, much of the code is decentralized from the project and accessed whenever needed instead of being stored in a complicated structure of source files. Those imported functions later replace most of the loops from original implementation, which is shown on the *Figure 11*. Microsoft's implementation of `frodo_mul_add_sb_plus_e()` function and *Figure 12*. Implementation of `frodo_mul_add_sb_plus_e()` in Python below to show main difference between Pythonic way to handle matrix computations on one hand, and the same code written in C. Firstly, one of the functions from `frodo_macrify.c` is presented, which does multiply two vectors and adds third to it. This is computed inside

```

void frodo_mul_add_sb_plus_e(uint16_t *out, const uint16_t *b, const uint16_t *s, const uint16_t *e)
{ // Multiply by s on the left
  // Inputs: b (N x N_BAR), s (N_BAR x N), e (N_BAR x N_BAR)
  // Output: out = s*b + e (N_BAR x N_BAR)
  int i, j, k;

  for (k = 0; k < PARAMS_NBAR; k++)
  {
    for (i = 0; i < PARAMS_NBAR; i++)
    {
      out[k*PARAMS_NBAR + i] = e[k*PARAMS_NBAR + i];
      for (j = 0; j < PARAMS_N; j++)
      {
        out[k*PARAMS_NBAR + i] += s[k*PARAMS_N + j] * b[j*PARAMS_NBAR + i];
      }
      out[k*PARAMS_NBAR + i] = (uint32_t)(out[k*PARAMS_NBAR + i]) & ((1<<PARAMS_LOGQ)-1);
    }
  }
}

```

Figure 11. Microsoft's implementation of `frodo_mul_add_sb_plus_e()` function (Microsoft, 2019)

a triple-nested loop and, as straightforward as it is, it shows all the details of implementation of this computation. Python implementation of the same function, as shown on picture below, hides all the logic under functions that operate on entire vectors, not just their single elements. It is possible to re-write the code from C to Python with all these loops instead of functions, but it is certainly not a Pythonic way for this task, and it would result in a much slower execution comparing to the same code that uses functions, not loops. All the loops are of course inside these functions and it is surely better to let Python take care of them. Python version might seem a little overwhelming at first, with all these nested functions and ranges but it is only before one takes a closer look at official documentation of these functions, which clearly explains what arguments they take and what output is expected. This means that a programmer does no longer care about how i.e. matrix multiplication is computed – but only what to put inside it. Also, one may notice that since it is just matrix multiplication and addition, it should be much shorter in Python, calling only i.e. `multiply()` and `sum()` functions. However, as this is not standard matrix multiplication but a slightly modified version, it could not be therefore simplified more. This is usually the case in many other functions of Microsoft's FrodoKEM implementation.

One last thing to mention in the subject of differences between these implementations are Python decorators. Decorator is simply a function that takes a function as a parameter and returns a function.

```

@staticmethod
def mul_add_sb_plus_e(pm, out, b, s, e):
    # Reference params for shorter lines
    pr_n = pm.PARAMS_N
    pr_nb = pm.PARAMS_NBAR
    pr_lq = pm.PARAMS_LOGQ

    # Calculate how many elements from s take, knowing that it should have size 40960
    # which is 5120 * 8 => meaning to be able to divide it into proper sub-vectors
    s_range = (pr_n * pr_nb * (pr_nb)) // pr_nb

    # Take range of elements from e vector
    out[:pr_nb**2 + pr_nb] = e[:pr_nb**2 + pr_nb]

    # array_split divides into n vectors, but what if we want to have vectors of n
    # len, and as many of them as array_split could divide vector into? Thus use of ceil(...)
    s_vec = hstack(tile(array_split(s[s_range:], ceil(s_range/pr_n)), pr_nb))
    # Split b vector into pr_n vectors and transpose them ('F').
    b_vec = tile(array(split(b, pr_n)).flatten('F')[:pr_n*pr_nb], pr_nb)

    out[:pr_nb**2 + pr_nb] += sum(array(split(s_vec * b_vec, pr_nb**2)), axis=1)
    out[:pr_nb**2 + pr_nb] = bitwise_and(out[:pr_nb**2 + pr_nb], ((1 << pr_lq) - 1))
#

```

Figure 12. Implementation of `frodo_mul_add_sb_plus_e()` in Python

“Decorating” functions is done using “@”. On the picture with Python code shown before there is decorator called “@staticmethod”. It is used to indicate that specific method in a class is only generated once and therefore reside in one specific place in memory. As classes in Python does not provide mechanisms for creating static methods, encapsulating them or class’ parameters etc. these functionalities can be used through decorators. Another common use for decorators is to define what specific parameters does function/method accept, what types of parameters and what it returns.

3.5.2. Fail rate of Python implementation

There is one caveat that comes with this implementation in Python. Once for a time the encapsulation-computed shared secret vector does not equal the one generated by decapsulation. This issue was thoroughly tested and compared with results of Microsoft’s implementation. The only source of uncertainty in the key generation is one randomly generated vector at the beginning of the algorithm, and there is similar generation as well in encapsulation algorithm. Decapsulation algorithm is completely deterministic. The problem is, for some of these mentioned randomized vectors the issue persists (only in Python implementation). It was not possible to find the source of the issue. In order to picture this with concrete numbers, the following tests were run with command:

```
# python3 -m Application --test --fails --number 1000
```



```
=====76.4
Testing correctness of key encapsulation mechanism (KEM), system FrodoKEM-640, tests for 1000 iterations
=====
Time started: 2019-12-15 15:26:53.844436
Estimated time: 0 seconds
Current operation: keyGeneration
Current iteration: 1000
Correct results: 998
Incorrect results: 2
[#####] (100.0%)
Tests finished.

Time finished: 2019-12-15 20:52:22.189780
Tests have taken 19528.35 seconds
```

Figure 13. Testing fail rate of FrodoKEM Python implementation

The Application module was run with specific options in order to deliver results of tests run 1000 times – key generation, encapsulation, decapsulation and comparison of computed shared secret vectors. There were only 2 incorrect results (mismatch) and 998 correct – failure rate is extremely small. The Python script shows current operation, current iteration (with a progress bar below) and based on n-1 iteration (starting iteration nr. n=1) it approximates estimated run time. The tests run 19528.35 seconds (~ 330 mins => 5,5 hrs).

3.5. Demonstration application

Main purpose of this dissertation is analysis of post quantum key exchange protocols. There was a need to create a simple, command-line tool that would automate testing process. It was also written in Python, and some of its features are pictured on figures in previous section, and *Efficiency tests* section as well. This section will provide a brief overview of the most important features and navigation through options available in that command line tool.

Help in the tool can be displayed by simply executing it in terminal, without any arguments:

```
# python3 -m Application
```

```

cheshiecat@DESKTOP-83B1GVP:/mnt/c/Users/przem/OneDrive/PycharmProjects/AgainstQuantum
$ p3 -m Application
usage: Application [-h] [-p P] [-a A] [-l] [--svconn SVCONN] [-c] [-f F]
                  [--clport CLPORT] [--nick NICK] [--secure]
                  [--mode {visual,text}] [--test] [--ops OPS] [--fails]
                  [--repeat REPEAT] [--number NUMBER] [--system SYSTEM]

Chat and key exchange testing

optional arguments:
  -h, --help            show this help message and exit

client-server basic options:
  -p P                  connection port
  -a A                  connection address
  -l                    start server
  --svconn SVCONN       number of clients that can be connected to the server
                        at the time
  -c                    connect to a server
  -f F                  send file to the server - provide path to the file

chat options:
  --clport CLPORT       change port on which client will listen for server's
                        response (default 9777)
  --nick NICK           change default-random nick
  --secure               starts secure connection
  --mode {visual,text}  display mode

testing key exchange:
  --test                time benchmark of [operations]. Default is testing of
                        all operations
  --ops OPS             possible operations are: keyGeneration, encapsulation,
                        decapsulation
  --fails               test only if encapsulation matches decapsulation,time
                        benchmark is not the goal
  --repeat REPEAT       How many times to repeat time measurement
  --number NUMBER       How many times execute specified operations
  --system SYSTEM       Choose FrodoKEM version (Frodo640, Frodo976,
                        Frodo1344)

```

Figure 14. Available command-line options for the application

Besides tests, which are discussed in other sections, it is also possible to start a simple chat (with either text or visual mode). The purpose of this feature is the presentation of Python's FrodoKEM implementation in real-world scenario. The tool has a `--secure` option which starts secure connection with the server to exchange messages or ensures that server accepts only secure connections (when used with the server). Chat in the visual mode is presented on the Figure 15. Chat in visual mode - client for the client (for the server in visual mode window looks similarly, except that there is no box for currently edited message by user). Chat in the text mode is presented on the Figure 16. Chat in the text mode – client for the client, and on the Figure 17. Chat in the text mode - server for the server.

Chat was run with the following options for the server:

```
# python3 -m Application -l --mode visual
```

And the following options for the client

```
# python3 -m Application -c --mode visual
```

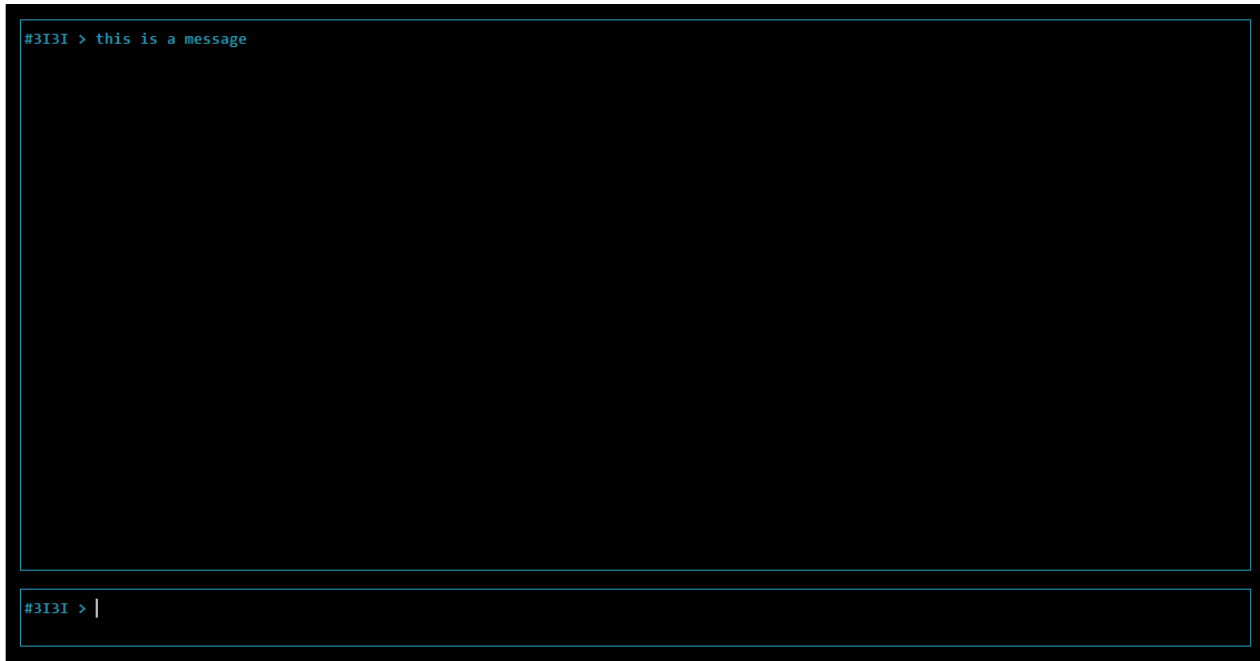


Figure 15. Chat in visual mode - client

Logic of this application is as follows: server is started at a specific port and address, either in the visual or text mode. In visual mode, only messages sent to the server are displayed in the window, and all logs are sent to the log file residing in the same directory as server. The same is for client – either visual or text mode, in the former logs go to the log file, in the latter – they are printed on the std output. Server accepts specific number of connections, which can be set with an option. It has a function of “chat room” – clients connect to it, send messages, and every message server received is sent to all connected clients, except the sender.

```
cheshiecat@DESKTOP-83B1GVP:/mnt/c/Users/przem/OneDrive/PycharmProjects/AgainstQuantum
$ p3 -m Application -c --mode text
[*] Connection with the server is insecure.
[*] Listening for messages from server on 0.0.0.0:9777

#544L > this is a message
[*] Sending message returned: {"code": 200}

#544L >
```

Figure 16. Chat in the text mode – client

When client starts, first thing that user sees is that 4-letter-number string enclosed within prompt. It is the user's nick, which is randomly chosen by default, and can be set to specific value in command-line options. After connecting to the server, client starts a separate thread on a specific port that listens for any incoming messages from the server. This behavior is similar to push & poll mechanism from observer pattern (Okhravi, n.d.)

```
cheshiecat@DESKTOP-83B1GVP:/mnt/c/Users/przem/OneDrive/PycharmProjects/AgainstQuantum
$ p3 -m Application -l --mode text
[*] Listening on 0.0.0.0:9999
[*] Accepted connection from: 127.0.0.1:49722
[*] New sender added: 127.0.0.1:9777
```

Figure 17. Chat in the text mode - server

Application is much more interesting under the hood. Curses library was used to draw in console window (Python curses, n.d.), which is basically not the good news, because it is a very low-level library taken straight from C. Thus, functions which i.e. get message from user, or display it in the chat window, are much more complicated than it was planned. Getting message is checking character by character passed, and all additional actions (i.e. confirming message, back-spacing message etc.) are just a cascade of ifs. Class that is responsible for this behavior is called ChatManager. The other class, which is responsible for connection is ConnectionManager. ChatManager is the main class instantiated in __main__.py file (which is one of first files executed when python loads library with -m switch). ConnectionManager is instantiated within the

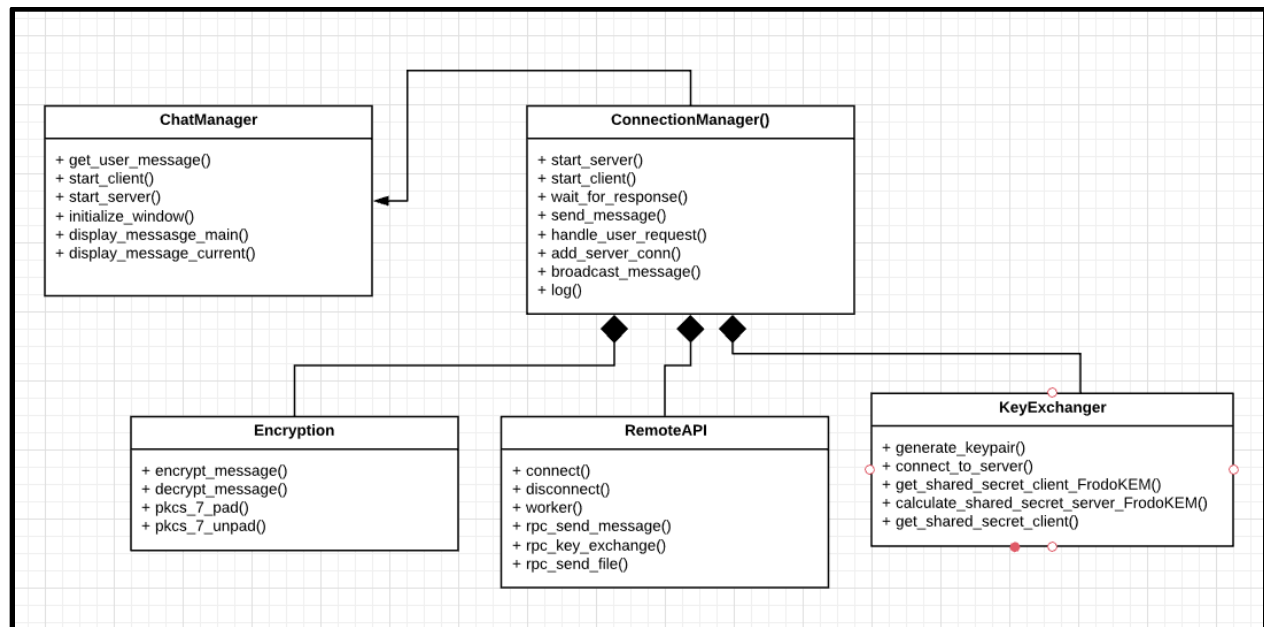


Figure 18. General overview of classes used in application

ChatManager class. Hierarchy of these classes in this application is shown on the Figure 18. General overview of classes used in application. ConnectionManager class has a few internal classes. First one – RemoteAPI – is responsible for providing functions that client can operate on when connecting to the server. It is the RPC (Remote Procedure Call) interface, or a set of functions and their parameters, which allow client to simply instantiate this class, and connect to the server through a standardized object. The other

internal class is KeyExchanger. It is responsible for secure key exchanging in order to secure connection. Default and only implemented key exchange is FrodoKEM, which is LWEKE scheme-based (Bhattacharya) (Joppe Bos), but extracting these functions to an external class allow effortless replacing/adding some other key exchange protocol in the future. The last internal class is Encryption. After the keys have been exchanged, Encryption provides function to encipher and decipher messages using 16-byte shared secret as the key, and AES with ECB mode as the cipher.

Last thing to say is that encrypted connection still has some issues. Mainly, due to the non-deterministic characteristics of FrodoKEM, key exchange fails sometimes which results in server and client generating different shared secret from the same pair of keys. There is also a problem with connecting more than one client to the server. If the key exchange ends with success, despite many tries another client is not able to successfully generate shared secret when connecting to the server. However, this issue does not make the presentation of FrodoKEM in this application impossible, so investigation of this problem is beyond the scope of this dissertation.

3.6. Existing libraries

This section will provide information about existing implementations of both FrodoKEM and SIDH (The latter will be discussed firstly). Most important place on the Internet to start with is Open Quantum Safe (OQS) project (OQS, 2019) which aims at support of quantum-resistant cryptography development through research, prototyping and evaluation of NIST quantum-safe candidates. Currently, OQS GitHub repository contains both mentioned protocols, as well as a number of other algorithms. Most of these implementations are part of several Microsoft repositories, and are written mainly in C, sometimes supplemented by Assembler patches to support bottlenecks. However, this project encourages everyone to support its development, as far as contribution's security and efficiency meets NIST criteria. Finally, due to low-level implementation of most of algorithms, OQS library is buildable on various platforms, such as Windows, Linux, BSD and is also accessible on portable architectures – ARM.

Another repository that should be considered in this chapter is Java implementation of SIDH, written by DeFeo Jao and Plut (Plut, n.d.). This implementation is mostly based on Microsoft's implementation of SIDH in C, and yet these authors extended original code by rewriting it in compliance with OOP (Object-Oriented Programming). As such, this new implementation of SIDH is probably not the most efficient one, but it is certainly easier to compile, run, test or analyze the code.

3.7.1. SIDH implementation in Java by DeFeo Jao and Plut

Main function in this implementation starts with creating pair of keys - for both parties asymmetrically. Next up there is key exchange, as well as some efficiency tests is located in *SidhTest.java*. The code is poorly commented, so sometimes it is a struggle to understand programmers' intentions. On the other side it is simple and concise at the very top level, so that without getting into detail it resembles standard Diffie-Hellman (DH) algorithm (save the non-symmetry – which is directly associated with structure of the SIDH. This matter is sufficiently explained in Theory section, but in a simple word both sides of key exchange have to calculate public keys in a slightly different manner, as opposed to traditional DH exchange).

At a first glance, in this implementation random numbers that are fundamental for security of key exchange are hard-coded inside main function. This saves time of execution, but it should be pointed out that currently SIDH implementations that are being tested in the real-world scenarios have ephemeral keys (as opposed to static keys) – which means that key exchange parameters are changing for each initialization of conversation between parties. This is the only option that is now considered secure against either classical cryptanalysis, as well as upcoming quantum-based one. This implementation consists of 7 other files, 4 of them directly associated with DH – *SidhKeyExchange.java*, *SidhKeyPair.java*, *SidhPrivateKey.java*, *SidhPublicKey.java*, and three others that are more related to under-the-hood computations in this specific variation of DH: *MontCurve.java*, *Fp.java*, *EcIsogeny.java*. Much of the code is, simply put, a copy of Microsoft's implementation. Since implementation of SIDH is not the subject of this dissertation, this Java implementation will not be further analyzed.

3.7.2. FrodoKEM implementation in plain C by Microsoft

As mentioned, there is so far only one FrodoKEM implementation and it is written by Microsoft (used in OQS library). It is very efficient – many operations i.e. matrix multiplications are written in a way that

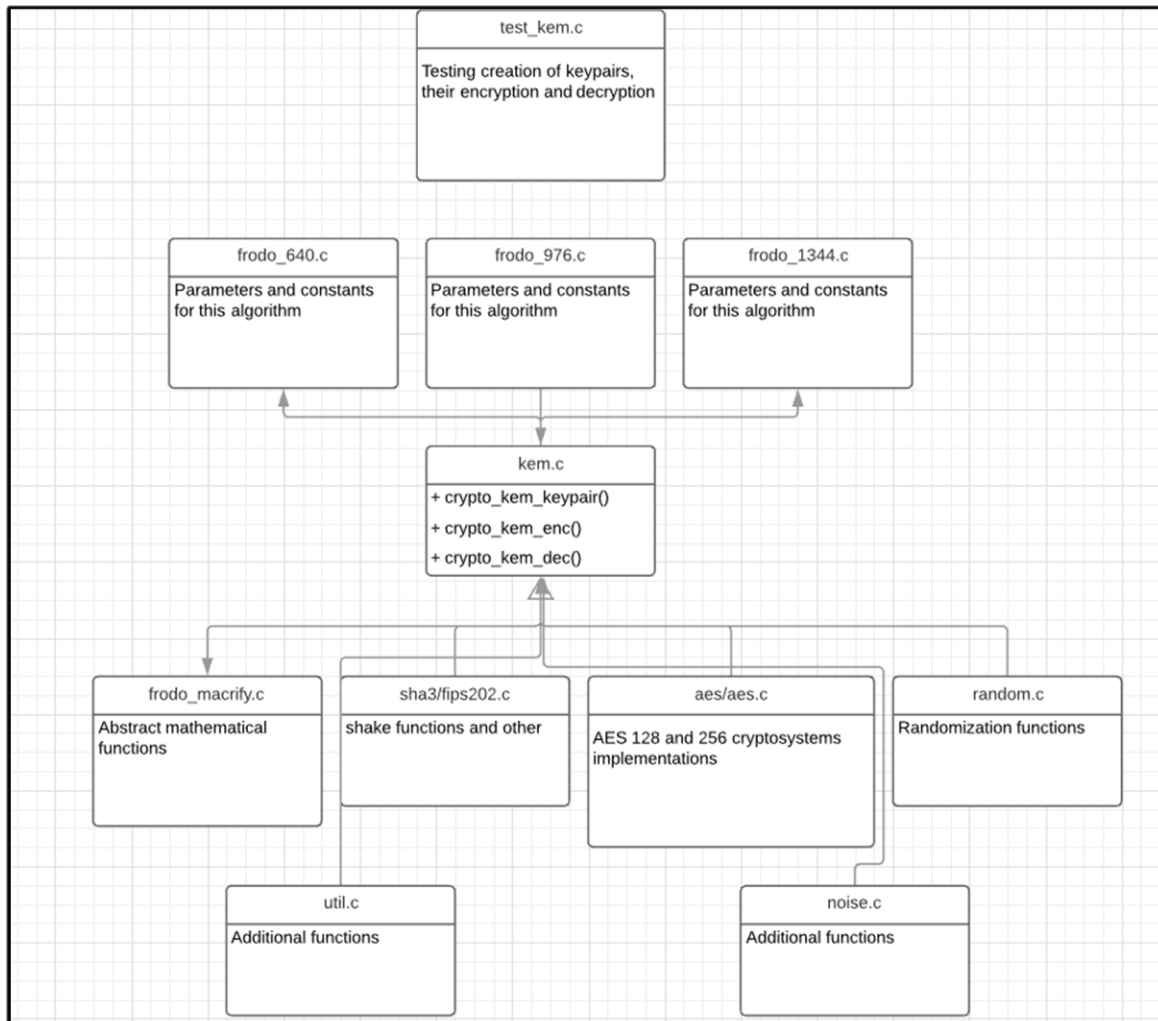


Figure 19. General overview of classes in Microsoft's implementation (based on (Microsoft, 2019))

operations are going on multiple rows simultaneously – thus reducing number of loop's iterations and total time of computation. Some functions implement AVX (Intel's Advanced Vector Extensions) additionally improving computation's speed. This implementation is also universal – it has different functions for different Operating Systems (i.e. Windows, Linux, BSD), CPU architectures (i.e. for ARM) and even different byte orders (some functions require to translate data stored in Big Endian into Little Endian). There are a few functions borrowed from other projects – majority of the code is tailored to the needs of this algorithm. This is the greatest advantage of low-level programming – many functions can be implemented from a scratch giving programmers ability to write a code that will exactly suit their needs. It will never be the fastest way to go, albeit it would be the best way for improving efficiency.

Structure of Microsoft's implementation is straightforward. There are three versions of the algorithm – 640, 976 and 1344. They differ mostly in length of generated keys, but also sometimes use slightly different functions. As shown on the Figure 19. General overview of classes in Microsoft's implementation (based on), there are three files at the top level of diagram that define parameters for each version. Based on which version user has chosen, one of these top-level API functions later calls one of deeper functions in kem.c file. Please note that this is a simplified structure of files in this implementation – all files with extension *.h are omitted and a few other configuration files that are not relevant to this analysis. Eventually, all

functions for either generating, encrypting or decrypting keys depend on a few bottom-level source files that implement all the algorithm logic: from randomization, through abstract mathematical computations on matrices, to shake and AES-related functions. These files stand for about 80% LOC (Lines Of Code) in this project. Most of them are highly specialized, tailored functions written just for the needs of FrodoKEM, whereas the other came from various GitHub repositories, each time providing links for the original implementation. The user is only given a bunch of test files, one of them called kem_test.c, which provide easy to use and analyze tests of efficiency for FrodoKEM. The usage and general description of how to use the program is available on the project's GitHub page (Microsoft, 2019).

3.7.3. SIDH implementation in plain C by Microsoft

This section is only to mark that Microsoft has already implemented SIDH algorithm as a part of OQS project and it will not be further discussed as only FrodoKEM protocol was subject of thorough analysis and implementation in this dissertation. It is only worth to mention that much of the code used in FrodoKEM implementation was used in SIDH as well.

4. Analysis

This section is about presentation and analysis of efficiency tests on Microsoft's FrodoKEM implementation, and implementation in Python which is a product of this dissertation. Both tests were run for the FrodoKEM-640-AES version in the same environment which is (see Figure 20. OS and CPU specification (output for processor 1 was omitted) for details):

Kali Linux 5.3.0-kali1-amd64 on VirtualBox 6.0.14


```

root@kali:~# cat /proc/version
Linux version 5.3.0-kali1-amd64 (devel@kali.org) (gcc version 9.2.1 20191030 (Debian 9.2.1-16)) #1 SMP
Debian 5.3.7-1kali2 (2019-11-04)
root@kali:~# cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 60
model name     : Intel(R) Core(TM) i3-4160 CPU @ 3.60GHz
stepping       : 3
cpu MHz        : 3598.956
cache size     : 3072 KB
physical id    : 0
siblings       : 2
core id        : 0
cpu cores      : 2
apicid         : 0
initial apicid : 0
fpu            : yes
fpu_exception  : yes
cpuid level    : 13
wp             : yes
flags          : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx f
xsr sse sse2 ht syscall nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_known_f
req pni pclmulqdq ssse3 cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lah
f_lm abm invpcid_single pti fsgsbase avx2 invpcid flush_lld
bugs           : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs
bogomips       : 7197.91
clflush size   : 64
cache alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:

processor       : 1

```

Figure 20. OS and CPU specification (output for processor 1 was omitted)

Processor – Intel Core i3-4160 CPU @ 3.60 GHz, 4GB RAM DDR3 1333 MHz, (2/4 available CPUs were used in VirtualBox configuration)

4.1. Efficiency tests

First implementation that is going to be tested and analyzed in this section is Python implementation. The tests were executed using Python's Timeit library, which is popularly used to benchmark Python scripts. One thing that could be mentioned at the beginning is difference between number of iterations and number of repetitions (--number and --repeat switches, respectively).

Algorithm 10. Timeit time measurement pseudocode

```

samples = []
for _ in range(repeat):
    # start timer
    for _ in range(number):
        do_work()
    # end timer
    samples.append(duration)

```

Repeat is the number of samples to benchmark, and Number specifies the number of times to repeat the code for each sample. Simple pseudo-code that demonstrates this relation is on the *Algorithm 10*. Timeit time measurement pseudocode.

Most interesting switch in this case is `--repeat`, because it gives more specific information about time of execution. The tests were run with the command:

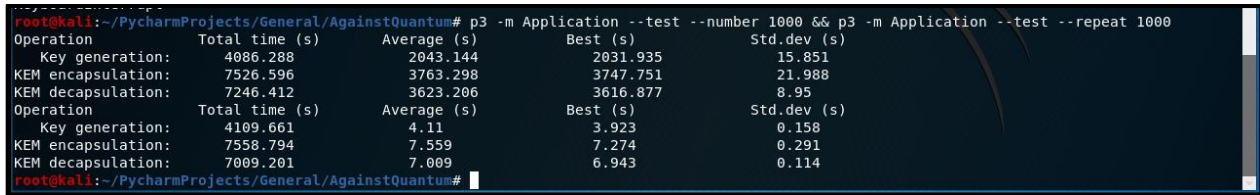
```
# python3 -m Application --test --repeat 1000
```

The results are presented in the *Table 9*. FrodoKEM Python implementation time benchmark for the sake of clarity (for original results, see *Figure 21*. FrodoKEM Python implementation time benchmark).

Operation	Total time (s)	Average (s)	Best (s)	Std. dev(s)
Key generation	4109.661	4.11	3.923	0.158
KEM encapsulation	7558.794	7.559	7.274	0.291
KEM decapsulation	7009.201	7.009	6.493	0.114

Table 9. FrodoKEM Python implementation time benchmark

Total time in the first column is the sum of all individual test executions of specific operation, average is a simple average of all results, Best column presents smallest interval the particular operation has taken, and std. dev is the standard deviation, or the statistic measure showing disperse of values between all results.



```
root@kali:~/PycharmProjects/General/AgainstQuantum# p3 -m Application --test --number 1000 && p3 -m Application --test --repeat 1000
Operation      Total time (s)  Average (s)  Best (s)  Std.dev (s)
Key generation: 4086.288       2043.144    2031.935   15.851
KEM encapsulation: 7526.596     3763.298    3747.751   21.988
KEM decapsulation: 7246.412     3623.206    3616.877    8.95
Operation      Total time (s)  Average (s)  Best (s)  Std.dev (s)
Key generation: 4109.661       4.11        3.923      0.158
KEM encapsulation: 7558.794     7.559       7.274      0.291
KEM decapsulation: 7009.201     7.009       6.943      0.114
root@kali:~/PycharmProjects/General/AgainstQuantum#
```

Figure 21. FrodoKEM Python implementation time benchmark

Microsoft's implementation was considerably faster. Before showing the results however, the source code had to be modified, changing *seconds* variable in the tests/test.kem file to value *1000*. *Figure 22*. Microsoft's FrodoKEM implementation – kem_bench() unmodified presents original kem_bench() function, and the *Figure 23*. Microsoft's FrodoKEM implementation – kem_bench() modified own modifications:

```

static void kem_bench(const int seconds)
{
    uint8_t pk[CRYPTO_PUBLICKEYBYTES];
    uint8_t sk[CRYPTO_SECRETKEYBYTES];
    uint8_t ss_encap[CRYPTO_BYTES], ss_decap[CRYPTO_BYTES];
    uint8_t ct[CRYPTO_CIPHTEXTBYTES];

    TIME_OPERATION_SECONDS({ crypto_kem_keypair(pk, sk); }, "Key generation", seconds);

    crypto_kem_keypair(pk, sk);
    TIME_OPERATION_SECONDS({ crypto_kem_enc(ct, ss_encap, pk); }, "KEM encapsulate", seconds);

    crypto_kem_enc(ct, ss_encap, pk);
    TIME_OPERATION_SECONDS({ crypto_kem_dec(ss_decap, ct, sk); }, "KEM decapsulate", seconds);
}

```

Figure 22. Microsoft's FrodoKEM implementation – kem_bench() unmodified

And the modified version:

```

static void kem_bench(const int seconds)
{
    uint8_t pk[CRYPTO_PUBLICKEYBYTES];
    uint8_t sk[CRYPTO_SECRETKEYBYTES];
    uint8_t ss_encap[CRYPTO_BYTES], ss_decap[CRYPTO_BYTES];
    uint8_t ct[CRYPTO_CIPHTEXTBYTES];

    TIME_OPERATION_ITERATIONS({ crypto_kem_keypair(pk, sk); }, "Key generation", 1000);

    crypto_kem_keypair(pk, sk);
    TIME_OPERATION_ITERATIONS({ crypto_kem_enc(ct, ss_encap, pk); }, "KEM encapsulate", 1000);

    crypto_kem_enc(ct, ss_encap, pk);
    TIME_OPERATION_ITERATIONS({ crypto_kem_dec(ss_decap, ct, sk); }, "KEM decapsulate", 1000);
}

```

Figure 23. Microsoft's FrodoKEM implementation – kem_bench() modified

The tests were run with the following command:

```
# cd ~/Desktop/Frodo-LWEKE
```

```
# make clean && make && clear && ./frodo640/test_KEM
```

Results delivered by Microsoft's implementation were considerably faster. As with the results presented in Python implementation, they were extracted in the *Table 10*. Microsoft's FrodoKEM implementation time benchmark below (for original results, see *Figure 24*. Microsoft's FrodoKEM implementation time benchmark).

Operation	Iterations	Total time (s)	Time (μs): mean	Pop. stdev	Cycles: mean	Pop. stdev

Key generation	1000	1.809	1809.253	1359.977	6508324	4894413
KEM encapsulation	1000	3.096	3095.817	3792.241	11139014	13648268
KEM decapsulation	1000	2.613	2613.078	2611.287	9399235	9397116

Table 10. Microsoft's FrodoKEM implementation time benchmark

Columns are similar to those in Python implementation, except for the CPU cycles. It was not possible to access them from any Python library as this is a higher abstraction layer than in plain C.

```

root@kali: ~/Desktop/PQCrypto-LWEKE
Operation      Iterations  Total time (s)  Time(us): mean  pop. stdev  Cycles: mean  pop. stdev
Key generation      1000        1.809          1809.253        1359.977    6508324       4894413
KEM encapsulate     1000        3.096          3095.817        3792.241    11139014      13648268
KEM decapsulate     1000        2.613          2613.078        2611.287    9399235       9397116
root@kali:~/Desktop/PQCrypto-LWEKE#

```

Figure 24. Microsoft's FrodoKEM implementation time benchmark

To give a better picture of the size of difference between these results, they were summarized in the *Table 11*. Summary of efficiency tests in both implementations. After some simple calculations, values in the parenthesis indicate that Python implementation is only about 2500x slower than C implementation. That is the real price for comfortability that gives Python.

Operation	Total time (s) - C	Mean time (s) - C (* 10⁻⁶)	Total time (s) - Python	Mean time (s) - Python
Key generation	1.809	0.00180	4109.661 (2271x ↓)	4.11 (2283x ↓)
KEM encapsulation	3.096	0.00309	7558.794 (2441x ↓)	7.559 (2446x ↓)
KEM decapsulation	2.613	0.00261	7009.201 (2682x ↓)	7.009 (2685x ↓)

Table 11. Summary of efficiency tests in both implementations

Microsoft has also provided efficiency tests of FrodoKEM in its documentation (Erdem Alkim, 2019). Instead of seconds, results were provided in cycles, not seconds. Performance (in thousands of cycles) of FrodoKEM on a 3.4GHz Intel Core i7-6700(Skylake) processor with matrix A generated using AES128. Results are reported using OpenSSL's AES implementation which exploit AES-NI instructions, with and without support of Intel's AVX. Cycle counts are rounded to the nearest 103 cycles. Results are presented in the *Table 12*. Microsoft's FrodoKEM implementation original time benchmark below, and the outcome for FrodoKEM-1344 version were provided as well for comparison. In order to convert these results to seconds, it is theoretically possible to divide cycle count by processor's speed (i.e. $1,384 / 3,4 * 10^9 = 4,0705 * 10^{-7}$), however these results will not be accurate, as processor can change its current frequency depending on several factors (i.e. temperature).

Scheme	KeyGen (cycles)	Encaps (cycles)	Decaps (cycles)	Total (Encaps + Decaps)(cycles)
FrodoKEM-640-AES	1,384	1,858	1,749	3, 607
FrodoKEM-1344-AES	4,756	5,981	5,748	11, 729
FrodoKEM-640-AES-AVX	1,388	1,879	1,768	3, 647

FrodoKEM-1344-AES-AVX	4,744	6,026	5,770	11, 796
-----------------------	-------	-------	-------	---------

Table 12. Microsoft's FrodoKEM implementation original time benchmark (Erdem Alkim, 2019)

4.2. Security tests

As mentioned in method section, presentation application allows to run chat in either *insecure* or *secure* mode. The latter does key exchange between client and server (using implemented FrodoKEM) and from that moment all communication is secured with AES (16-byte key – ECB mode).

In order to test this behavior, server is started on a specific port using command:

```
# python3 -m Application -l --mode text --secure
```

Next step is to start client and connect to the server:

```
# python3 -m Application -c --mode text --secure
```

Please note that client started without *--secure* option will not be able to connect to the “secure” server. Firstly, *Figure 25. Presentation application chat – client - insecure* will present client and *Figure 26. Presentation application chat - server – insecure connection* server connection on an insecure channel.

```
cheshiecat@DESKTOP-83B1GVP:/mnt/c/Users/przem/OneDrive/PycharmProjects/AgainstQuantum
$ p3 -m Application -c --mode text
[*] Connection with the server is insecure.
[*] Listening for messages from server on 0.0.0.0:9777

#M40D > this is a message
[*] Sending message returned: {"code": 200}

#M40D > |
```

Figure 25. Presentation application chat – client - insecure

```
cheshiecat@DESKTOP-83B1GVP:/mnt/c/Users/przem/OneDrive/PycharmProjects/AgainstQuantum
$ p3 -m Application -l --mode text
[*] Listening on 0.0.0.0:9999
[*] Accepted connection from: 127.0.0.1:50707
[*] New sender added: 127.0.0.1:9777

|
wsbridge.exe[64]:11420 α 191012[64]
```

Figure 26. Presentation application chat - server – insecure connection

Running Wireshark to sniff out this communication does successfully uncover what client has sent to the server (see *Figure 27. Sniffing plain-text communication between client and the server*)

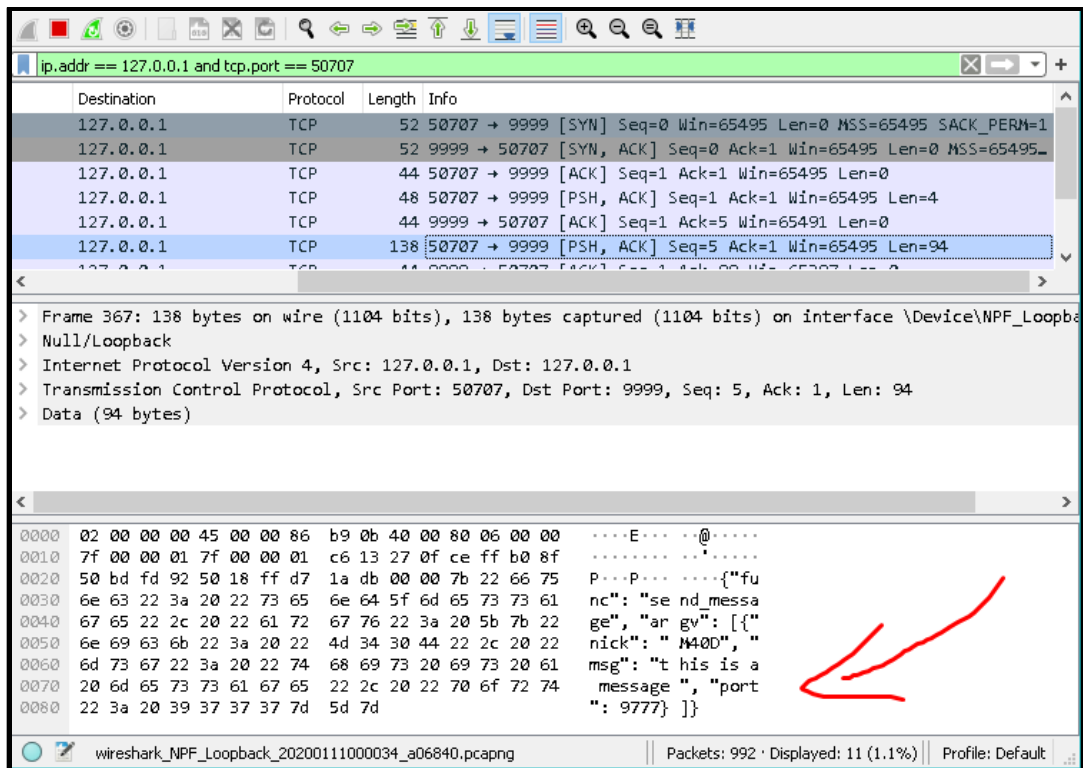


Figure 27. Sniffing plain-text communication between client and the server

Now that the unsecure communication has been analyzed, next step is to start both client and server with --secure option. Figure 28. Client connection with --secure option presents output from client window.

```

cheshiecat@DESKTOP-83B1GVP:/mnt/c/Users/przem/OneDrive/PycharmProjects/AgainstQuantum
$ p3 -m Application -c --mode text --secure
[*] Listening for messages from server on 0.0.0.0:9777
[*] Connection with the server is now secure.

#IYRI > this is a message
[*] Sending message returned: {"code": 200}

#IYRI > |
  
```

Figure 28. Client connection with --secure option

Client has informed user at the beginning that the connection is now secure. Client has sent a message. Meanwhile, output from server window does look much more interesting (see Figure 29. Server connection with --secure option). From logs it can be inferred that client has requested server's public key, server has accepted connection and after a successful key exchange secure connection starts with client.

```

cheshiecat@DESKTOP-8381GVP:/mnt/c/Users/przem/OneDrive/PycharmProjects/AgainstQuantum
$ p3 -m Application -l --mode text --secure
[*] Listening on 0.0.0.0:9999
[*] Accepted connection from: 127.0.0.1:51673
[*] Sending server's public key back to the client: <socket.socket fd=4, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 9999), raddr=('127.0.0.1', 51673)>
[*] Accepted connection from: 127.0.0.1:51674
[*] Connection secured with client: <socket.socket fd=5, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 9999), raddr=('127.0.0.1', 51674)>
[*] Accepted connection from: 127.0.0.1:51678
[*] New sender added: 127.0.0.1:9777

```

Figure 29. Server connection with --secure option

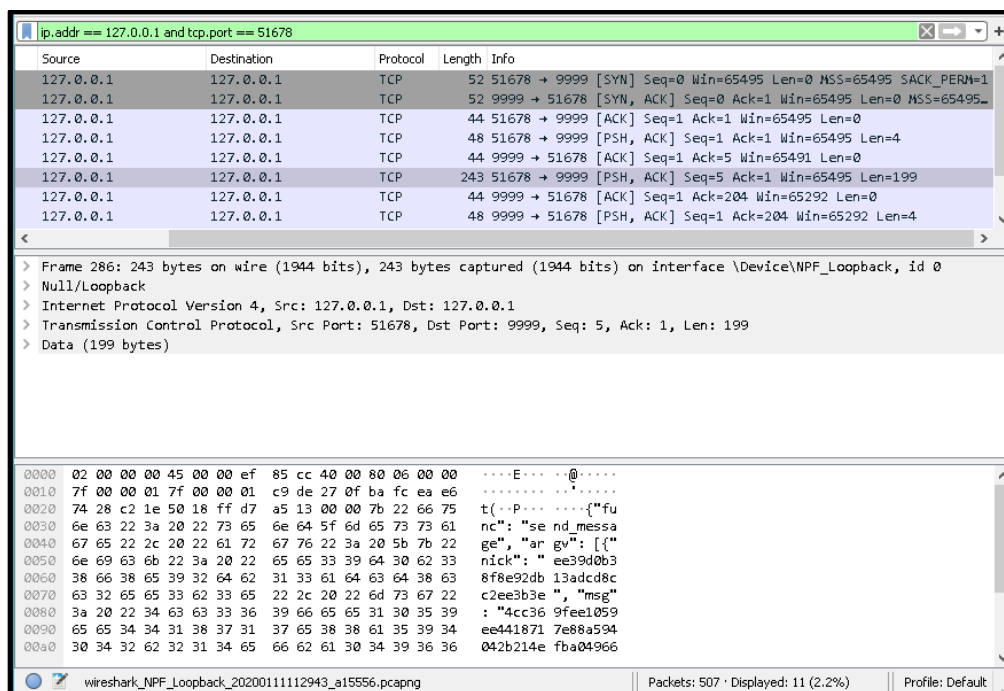


Figure 30. Wireshark sniffing secure connection

To make sure connection is secured, Wireshark is run again to sniff loopback interface. This time it was only possible to read function information, but all parameters, including message, are not in the plain-text form anymore (see Figure 30. Wireshark sniffing secure connection).

5. Summary

Main subject of this dissertation was analysis of post-quantum key exchange protocols, presentation of their structure and related mathematical problems – mainly calculating isogenies between elliptic curves (Isogeny problem) and solving a system of equations with probabilistic element (LWE problem). One of these protocols – FrodoKEM – was particularly analyzed and implemented using Python.

This implementation, as discussed in later chapters of this work, was about 2500x slower than original Microsoft's implementation, however, the code was significantly shorter and more abstract – giving the programmer a more comfortable interface to work with. It might seem straightforward to re-implement an algorithm based on provided source code in another common language (having similar syntax afterall). On the contrary, the re-implementation of FrodoKEM in Python was an effort-and-patience-demanding challenge that, being mistakenly perceived as simple at the beginning, has been ever since a complicated journey full of incompatibilities and language quirks, traveled with thorough, minute analysis and a bit of imagination. Abstract mathematical functions were the biggest challenge, in a sense that not only mathematical functions were sometimes beyond the reach of understanding, but these functions were implemented by Microsoft with several architectures in mind, therefore giving a complicated tree of decisions and non-standard instructions (such as Intel's AVX). Additionally, reimplementing four-nested loops from C to *no-visible-loops* objects in Python was also very demanding. From the beginning, the main goal of Python implementation was being as efficient as possible and secure FrodoKEM-640 targeting only CISC architecture (non-mobile applications). Security of FrodoKEM in Python is provided by most popular, approved libraries for secret pseudo-random number generation.

Demonstration application was implemented in order to give a tool to time-benchmark implemented FrodoKEM and picture one of its use-cases in a simple client-server chat application. This application allows connecting multiple clients to a server and send messages, which are later broadcasted by the server to other connected clients. It is also possible i.e. to send a file to the server. All these functionalities can be used in plain-text, therefore jeopardizing user privacy to any sniffing attacks, or secured with AES, which uses keys exchanged with FrodoKEM. Last sections of this dissertation demonstrate step-by-step analysis of communication in both cases, trying to sniff any messages using Wireshark.

Other subjects, such as the latest milestones in quantum computing, or basics of cryptography were also briefly discussed, in order to give the reader a better general perspective of problems discussed in this work. All in all, thorough analysis of mathematical problems related to algorithms discussed here, as well as Python implementation of FrodoKEM were the subject of this dissertation and hopefully will bring a great educational value to the readers not familiar with these subjects before. Python implementation of FrodoKEM is rather not a good candidate for NIST proposal because of its efficiency (comparing to C), however, it gives a good opportunity to understand the mechanisms of this algorithm, without having to interact with low-level, under-the-hood instructions.

6. Table of figures & tables

FIGURE 1. EXAMPLES OF LATTICES: FULL-RANK BASIS OF \mathbb{Z}^2 (LEFT) AND A NOT FULL RANK LATTICE BASED ON (REGEV O. , 2004).....	7
FIGURE 2. ECS ON A PLANE A) $Y^2 = X^3 - 3X + 3$ ($\Delta = 2160$) B) $Y^2 = X^3 + X$ ($\Delta = -64$) C) $Y^2 = X^3$ ($\Delta = 0$).....	8
FIGURE 3. SIMPLIFIED MODEL OF SYMMETRIC CRYPTOSYSTEM (STALLINGS).....	11
FIGURE 4. SIMPLIFIED MODEL OF ASYMMETRIC CRYPTOSYSTEM (STALLINGS)	12
FIGURE 5. PRESENTATION OF RSA ALGORITHM IN A GRAPHICAL FORM	14
FIGURE 6. PRESENTATION OF DH ALGORITHM IN A GRAPHICAL FORM	15
FIGURE 7. FUNDAMENTAL PARALLELEPIPED VISUAL REPRESENTATION (REGEV O. , 2004).....	18
FIGURE 8. EXAMPLE PRESENTATION OF SIDH KEY EXCHANGE USING CURVE25519 (BERNSTEIN)	34
FIGURE 9. FRODOKEM PYTHON IMPLEMENTATION CLASS STRUCTURE	36
FIGURE 10. IMPORTING LIBRARIES – FRODO_MACRIFY.PY	37
FIGURE 11. MICROSOFT’S IMPLEMENTATION OF FRODO_MUL_ADD_SB_PLUS_E() FUNCTION (MICROSOFT, 2019)	38
FIGURE 12. IMPLEMENTATION OF FRODO_MUL_ADD_SB_PLUS_E() IN PYTHON.....	39
FIGURE 13. TESTING FAIL RATE OF FRODOKEM PYTHON IMPLEMENTATION.....	40
FIGURE 14. AVAILABLE COMMAND-LINE OPTIONS FOR THE APPLICATION	41
FIGURE 15. CHAT IN VISUAL MODE - CLIENT	42
FIGURE 16. CHAT IN THE TEXT MODE – CLIENT	42
FIGURE 17. CHAT IN THE TEXT MODE - SERVER.....	43
FIGURE 18. GENERAL OVERVIEW OF CLASSES USED IN APPLICATION	43
FIGURE 19. GENERAL OVERVIEW OF CLASSES IN MICROSOFT’S IMPLEMENTATION (BASED ON (MICROSOFT, 2019)).....	46
FIGURE 20. OS AND CPU SPECIFICATION (OUTPUT FOR PROCESSOR 1 WAS OMITTED).....	48
FIGURE 21. FRODOKEM PYTHON IMPLEMENTATION TIME BENCHMARK	49
FIGURE 22. MICROSOFT’S FRODOKEM IMPLEMENTATION – KEM_BENCH() UNMODIFIED	50
FIGURE 23. MICROSOFT’S FRODOKEM IMPLEMENTATION – KEM_BENCH() MODIFIED.....	50
FIGURE 24. MICROSOFT’S FRODOKEM IMPLEMENTATION TIME BENCHMARK.....	51
FIGURE 25. PRESENTATION APPLICATION CHAT – CLIENT - INSECURE.....	52
FIGURE 26. PRESENTATION APPLICATION CHAT - SERVER – INSECURE CONNECTION	52
FIGURE 27. SNIFFING PLAIN-TEXT COMMUNICATION BETWEEN CLIENT AND THE SERVER	53
FIGURE 28. CLIENT CONNECTION WITH --SECURE OPTION.....	53
FIGURE 29. SERVER CONNECTION WITH --SECURE OPTION.....	54
FIGURE 30. WIRESHARK SNIFFING SECURE CONNECTION	54
TABLE 1. DEFINITION1 (LATTICE)	7
TABLE 2. DEFINITION2 (ELLIPTIC CURVE)	8
TABLE 3. COMPARISON OF MOST COMMON PUBLIC-KEY CRYPTOSYSTEMS (STALLINGS)	13
TABLE 4. SUMMARY OF QUANTUM COMPUTING IMPLICATIONS FOR CYBERSECURITY (NISTIR 8105, 2016)	16
TABLE 5. FRODOKEM PARAMETERS’ SPECIFIC VALUES (ERDEM ALKIM, 2019)	24
TABLE 6. FRODOKEM KEY SIZES IN BYTES.....	29
TABLE 7. FRODOKEM KEY PARAMETERS SIZES, RANGES AND APPROXIMATED SECURITY	29
TABLE 8. COMPARISON OF DH VARIATIONS.....	30
TABLE 9. FRODOKEM PYTHON IMPLEMENTATION TIME BENCHMARK.....	49

TABLE 10. MICROSOFT’S FRODOKEM IMPLEMENTATION TIME BENCHMARK	51
TABLE 11. SUMMARY OF EFFICIENCY TESTS IN BOTH IMPLEMENTATIONS	51
TABLE 12. MICROSOFT’S FRODOKEM IMPLEMENTATION ORIGINAL TIME BENCHMARK (ERDEM ALKIM, 2019)	52

7. References

- Alkim, E. (2019). *FrodoKEM*. Retrieved from <https://frodokem.org/files/FrodoKEM-specification-20190330.pdf>
- Apon, D. (2012). An intro to lattices and learning with errors.
- Avrim Blub, A. K. (2003). Noise-Tolerant Learning, the Parity Problem, and the Statistical Query Model.
- Bernstein, D. J. (n.d.). Curve25519: New Diffie-Hellman Speed Records.
- blperez1. (2018). *A Guide to Post-Quantum Cryptography*. Retrieved from <https://securityboulevard.com/2018/10/a-guide-to-post-quantum-cryptography/>
- Costello, C. (2010). An introduction to supersingular isogeny-based cryptography.
- Craig Costello, P. L. (n.d.). Efficient algorithms for supersingular isogeny Diffie-Hellman.
- Cruise, B. (n.d.). Diffie-hellman key exchange - Khan Academy.
- David Jao Luca De Feo. (n.d.). Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies.
- desmos. (n.d.). Retrieved from <https://www.desmos.com/calculator/ialhd71we3>
- Eiichiro Fujisaki, T. O. (1999). Secure Integration of Asymmetric and Symmetric Encryption Schemes.
- Erdem Alkim, J. W. (2019). FrodoKEM - Learning With Errors Key Encapsulation documentation.
- Frank Arute, K. A. (2019). Quantum supremacy using a programmable superconducting processor.
- Goldwasser, D. M. (2002). Complexity of Lattice Problems.
- Hazewinkel, M. (2001). *Encyclopedia of Mathematics*.
- Hellman, W. D. (n.d.). New Directions in Cryptography.
- Huseyin Hisil, C. C. (n.d.). A simple and compact algorithm for SIDH with arbitrary degree isogenies.
- Karbowsky, M. (2013). Podstawy Kryptografii.
- Khan Academy. (n.d.). *Quotient Of Functions*. Retrieved from <https://www.khanacademy.org/math/algebra-home/alg-functions/alg-combining-functions/v/quotient-of-functions>

Koblitz, N. (2007). *The uneasy relationship between mathematics and cryptography*.

Martin R. Albrecht, R. P. (2019). On the concrete hardness of Learning with Errors.

Micciancio, D. (2010). CSE206A Lattice Algorithms and Applications.

Micciancio, D. (2010). Introduction to Lattices.

Microsoft. (2019). Retrieved from <https://github.com/Microsoft/PQCrypto-LWEKE/>

Mitchell, J. (n.d.). Key Exchange Protocols.

NISTIR 8105. (2016). *Report on Post-Quantum Cryptography*.

NSA. (2015). Cryptography Today.

Oded Regev, D. S. (2004). Lattices in Computer Science.

Okhravi, C. (n.d.). *Observer Pattern – Design Patterns (ep 2)*. Retrieved from https://www.youtube.com/watch?v=_BpmfnqjgzQ

OQS. (2019). Retrieved from openquantumsafe.org

Peikert, C. (2016). *A Decade of Lattice Cryptography*.

Plut, D. J. (n.d.). Retrieved from github.com/Art3misOne/sidh

Python Library. (n.d.). Retrieved from <https://docs.python.org/3/library/secrets.html>

rafael pass, a. s. (2010). A Course in Cryptography.

Regev. (n.d.). The Learning with Errors Problem.

Regev, O. (2004). Lattices in Computer Science.

Renner, C. P. (2014). Cryptographic security of quantum key distribution.

Richard Lindner, C. P. (2011). Better Key Sizes (and Attacks) for LWE-Based Encryption.

Rogaway, M. A. (1999). Relations Among Notions of Security for Public-KeyEncryption Schemes.

Savage, N. (2019). Hands-On with Google’s Quantum Computer.

Shor., P. W. (1994). Algorithms for quantum computation: Discrete logarithms and factoring. In Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on, pages 124–134. IEEE, 1994.

Silverman, J. H. (2009). *The Arithmetic of Elliptic Curves*.

Singh, S. (n.d.). The Code Book.

Stallings, W. (n.d.). Cryptography and Network Security.

Steven D. Galbraith, a. F. (n.d.). *COMPUTATIONAL PROBLEMS IN SUPERSINGULAR ELLIPTIC CURVE ISOGENIES*.

Stubbs, R. (2018). *Quantum Computing and its Impact on Cryptography*.

TIOBE. (2019). Retrieved from <https://www.tiobe.com/tiobe-index/>

Wikipedia. (2019). Retrieved from https://en.wikipedia.org/wiki/Python_%28programming_language%29