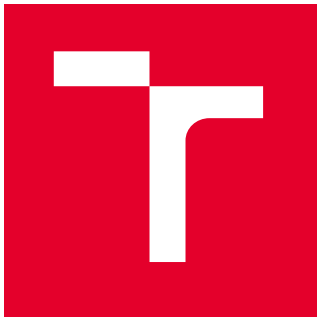

BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

QUANTUM AND POST-QUANTUM CRYPTOGRAPHY

KVANTOVÁ A POSTKVANTOVÁ KRYPTOGRAFIE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. Andrej Krivulčík

SUPERVISOR
VEDOUCÍ PRÁCE

doc. Ing. Jan Hajný, Ph.D.

BRNO 2022

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Information Security**

Department of Telecommunications

Student: Bc. Andrej Krivulčík ***ID:*** 203414

Year of study: 2 ***Academic year:*** 2021/22

TITLE OF THESIS:

Quantum and Post-quantum Cryptography

INSTRUCTION:

The topic is focused on the analysis of existing technologies for quantum and post-quantum key establishment and their combination. In the thesis, student will select suitable solutions and will propose the method for their combination. The system will be implemented as a demonstrator for the key establishment on the Linux platform. The main result of the thesis will be realized as a software demonstrator with postquantum libraries and HTTPS communication interface allowing downloading a key from the lab infrastructure. Demonstrator will use the hybrid key for AES 256 data encryption and will transfer the encrypted data over the network.

RECOMMENDED LITERATURE:

[1] NIST. Post-Quantum Cryptography PQC: Round 3 Submissions. [cit. 2021-09-09]. Dostupné z: https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[2] ETSI. ETSI GS QKD 014 V1.1.1 (2019-02): Quantum Key Distribution (QKD): Protocol and data format of REST-based key delivery API. [cit. 2021-09-09]. Dostupné z: https://www.etsi.org/deliver/etsi_gs/QKD/001_099/014/01.01.01_60/gs_QKD014v010101p.pdf

Date of project specification: 7.2.2022 ***Deadline for submission:*** 24.5.2022

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

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

ABSTRACT

With advances in quantum computing comes the threat of breaking the algorithms that are used in everyday communication. With this, an industry of post-quantum cryptography has emerged that develops algorithms resistant to quantum computers. The aim of this thesis is to study methods for combining and using keys established by quantum and post-quantum algorithms in such a way that if one of the given algorithms is broken the resulting hybrid key will still be secure. The resulting key is then used in encrypting the file using AES–256 which is sent between clients.

KEYWORDS

Key combination, Hybrid key, Key derivation, QKD, KEM, PQC, HTTPS, Python, NIST, AES, KMAC

ABSTRAKT

S pokrokom v rozvoji kvantových počítačov prichádza hrozba prelomenia algoritmov ktoré sa používajú pri bežnej komunikácii. S týmto vzniklo odvetvie pre post-kvantovú kryptografiu ktoré vyvíja algoritmy odolné voči kvantovým počítačom. Cieľom tejto diplomovej práce naštudovanie metód pre kombináciu a využitie kľúčov ustanovených pomocou kvantových a post-kvantových algoritmov takým spôsobom, keby pri došlo k prelomeniu jedného z daných algoritmov tak výsledný hybridný kľúč bude stále bezpečný. Výsledný kľúč je následne využitý pri šifrovaní súboru pomocou AES–256 ktorý je zaslaný medzi klientami.

KĽÚČOVÉ SLOVÁ

Kombinácia kľúčov, Hybridný kľúč, Derivácia kľúčov, QKD, KEM, PQC, HTTPS, Python, NIST, AES, KMAC

ROZŠÍRENÝ ABSTRAKT

Teoretická časť práce predstavuje kvantovú kryptografiu, jej kvantovú distribúciu kľúčov (QKD) a jeden z najpopulárnejších protokolov BB84 a koherentný jednosmerný protokol (COW). Postkvantová kryptografia predstavuje jej rôzne prístupy, kryptografiu založenú na mriežkach a problémy na nich, kryptografiu založenú na hashovacích funkciach a kryptografiu založenú na opravných kódoch. Organizácia NIST založila súťaž pri snahe o štandardizáciu post–kvantových algoritmov. Aktuálne táto sútaž je v treťom a zároveň finálnom kole. Protokoly sú podrobne opísané v ich príslušných kapitolách. Sú to dlhoročné protokoly, ako napríklad McEliece, alebo nové protokoly, ako napríklad NTRU alebo SABER. Každý algoritmus je založený na ich príslušných postkvantových problémoch, čím si navzájom dobre konkurujú.

Hlavným cieľom tejto práce je kombinácia dvoch kľúčov, kvantového a postkvantového, a ich bezpečné spojenie do jedného kľúča tak, aby v prípade oslabenia alebo prelomenia jedného z protokolov, druhý protokol udržal kľúč bezpečným až do momentu, kedy by bolo možné vykonať protiopatrenia.

Prvou z možných kombinácii a vytvorení hybridného kľúču je funkcia XOR. Funkcia XOR, zvaná aj ako perfektná šifra, sa na prvý pohľad javí ako implementačne jednoduchá funkcia, ktorá bude vo výsledku bezpečná. Ale opak je pravdou. Funkcia XOR má jednu veľmi podstatnú vlastnosť, a tou je reverznosť. Pri použití 2 kľúčov, A a B, funcia vráti kľúč C. Ale pri opätovnom použití kľúču C a jedného z kľúčov, napríklad A, funkcia vráti kľúč B. Táto vlastnosť je vo fundamentálnom rozpore s ideou tejto práce a preto je táto funkcia veľmi nedoporučovaná.

Hashovacie funkcie, na rozdiel of funkcie XOR, sú jednostranné. To znamená, že pri správne navrhnutej hashovacej funkcí nie je možné z výstupu zistit vstup. V dnešnej dobe sú hashovacie funkcie optimalizované na takej úrovni, kde výpočty trvajú iba niekoľko milisekúnd. Avšak je podstatné používať dodatočné parametre ako soľ, ktorá ešte viac znáhodní výsledný hash. Aj pri vlastnostiach ako sú odolnosť voči kolíziam, je hash funkcia, konkrétne rodina hash funkcii SHA–2 a SHA–3, stále veľmi obľúbená a používaná funkcia. Jedinou nevýhodou statická dĺžka výstupu.

Metódy kombinácie pomocou odvodzovania kľúču sú najviac pokročilé metódy a to z dôvodu, že kombinujú dobré vlastnosti hash funkcii s dodatočnou bezpečnostou a to aj v prípade kedy funkcie ako Extract–and–Expand Key Derivation Function (HKDF) a KECCAK Message Authentication Code (KMAC) sú pomalšie. Tieto funkcie umožňujú flexibilnú dĺžku výstupného kľúču.

Implementácia už spomínanej kombinácie dvoch kľúčov je písaná v programovacom jazyku Python. Simulácia klientov bola zriadená v internej sieti na Vysokom učení technickom v Brne (VUT). Klienti boli virtualizovaní v internom laboratóriu. Knižnica *PQcrypto* sa použila na dohodnutie postkvantového kľúča, pričom kvan-

tový kľúč bol stiahnutý z QKD serverou pomocou rozhrania HTTPS. Tento kľúč sa vyjednával prostredníctvom kvantového kanála s opísanými vlastnosťami. Výsledný skript kombinuje kľúče tromi rôznymi metódami, ktorými sú neodporúčaná funkcia XOR, ktorá je veľmi náchyná na útok pomocou kryptoanalýzi, kombinácia pomocou hash funkcie SHA–2 alebo SHA–3, alebo použitie pokročilej metódy na odvodenie kľúča KMAC. Metóda na odvodtovanie kľúču KMAC je najbezpečnejšia ale aj zároveň najpomalšia. Okrem samotnej kombinácie sa tajný súbor zašifruje hybridným kľúčom. Súbor, ktorý si užívateľ vyberie, je zašifrovaný výsledným hybridným kľúčom pomocou symetrickej šifry AES–256 v móde Galois/Counter (GCM) a odošlaný prostredníctvom TCP spojenia druhému klientovi, ktorý tento súbor dešifruje im vytvoreným hybridný kľúčom a uloží. Výstupná konzolová aplikácia je bez grafického používateľského rozhrania, ktorú možno spustiť s rôznymi počiatočnými riadiacimi argumentmi. Pripojená je aj používateľská príručka na jednoduchšiu navigáciu a spustenie uvedeného demonštrátora.

KRIVULČÍK, Andrej. *Quantum and Post-quantum Cryptography*. Brno: Brno University of Technology, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2022, 61 p. Master's Thesis. Advised by doc. Ing. Jan Hajný, Ph.D.

Author's Declaration

Author:	Bc. Andrej Krivulčík
Author's ID:	203414
Paper type:	Master's Thesis
Academic year:	2021/22
Topic:	Quantum and Post-quantum Cryptography

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

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

Brno ....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I would like to thank the supervisor of my master thesis doc. Ing. Ján Hajný, Ph.D. for his professional guidance, consultation, patience and suggestions for this work. I would also like to thank Dr. Alex Bryne for his mentoring and Luke Joyce for his moral support.

Contents

List of Figures

List of Tables

Listings

Introduction

One of the most fundamental mathematical problems that current cryptography is based on is the factorisation of two prime numbers and the discreet logarithm problem. Protocols like Diffie–Hellman key exchange (DH) and Digital Signature Algorithm (DSA) are bound on these mathematical problems. With progress in the current computation era, the rise of quantum computers would make these protocols obsolete and easily breakable. Thanks to the use of Shor's algorithm which could solve these mathematical problems in a polynomial time on a strong enough quantum computer.

With this problem in mind, a new cryptography branch was created called post-quantum cryptography (also called quantum-resistant cryptography). The main goal is to create quantum-resistant protocols that would not be solvable in polynomial time on quantum computers. Currently, there are 4 most popular categories of post-quantum cryptography, Lattice–based, Multivariate, Hash–based and Code–based cryptography. Each of these approaches solves the same problem by a different approach.

NIST (National Institute for Standards and Technologies) initiated a competition to evaluate possible candidates for standardisation in post-quantum cryptography. Currently, it's in the third round of submissions with algorithms such as McEliece, Crystal-Kyber, NTRU and SABER for post-quantum secret key sharing.

Furthermore, a difference between possible key combinations is described, due to the fact, that either quantum or post-quantum cryptography protocol can be broken at any given time. This brings danger to communication as a whole. By combining two established keys into one hybrid key, one for post–quantum cryptography and one for quantum cryptography proper way, we should achieve a key strong enough that, when one of the protocols is broken, the other can still hold the whole key secure. This key combination should be irreversible without the knowledge of both keys.

Main goal and thus practical implementation in chapter 5 will be described in programming language Python. Two test clients are set in virtual environment in internal BUT laboratory. Both clients are connected to quantum servers from Swiss manufacturer ID Quantique. Both quantum and post–quantum keys will be combined by all described methods and analysed. Output will be a functioning demonstrator for creating said hybrid key.

1 Quantum cryptography

Over the past few years, a quantum era has been unfolding. With the development of quantum computers, we will have access to an unprecedented amount of processing and computational power. Unfortunately, this new degree of power creates new problems to resolve. Most cryptography protocols that we are currently using could be broken with the use of Shor's algorithm, which could solve mathematical problems tied to these protocols in polynomial time. The prime factorisation and discreet algorithm problem, alongside its alternative on Elliptic curves, would be rendered unsecure in an era of quantum computing.

1.1 Quantum computing

Modern computers use a unit of information known as a bit, which can be stored as either logical 0 or 1. However, rather than using electrical charge on transistors to represent logical bits, quantum computers use a quantum principle called quantum superposition. In a quantum system, entities can have several quantum states. Whenever the entity exists, it could have several quantum states at once, sometimes all possible states. Only after the entity is observed can we determine its quantum state. This phenomenon is a building block for quantum mechanics. Schrödinger's Cat is a popular thought experiment based on quantum superposition. By locking a cat into a box and leaving it be, after some time, the same cat can have two possible states. Either the cat is dead or alive. Quantum mechanics tells us that cat has both states at once, that it is dead and alive at the same time. It is only when we measure its state (opening the box and observing the cat) that we can determine its current state [1].

For quantum computers, we use electrons as an entity and their spin as a state. Electrons can spin two ways: up or down. So instead of assuming a logical 0 or 1, it can possess a spectrum of values. By observing (measuring) the spin of an electron, we force it to collapse into the state of either 0 or 1. The outcome of the measurement is a probability vector, where the probability of one state is 100 % and the rest of the states are 0 %. Therefore, with the use of a probability vector instead of regular logical values, a quantum computer can parallelise these computations and offer a huge increase in computational power [1].

The next significant property of quantum computers is quantum entanglement. This is a phenomenon where two particles, in our case electrons, are entangled together, thus one is influencing the other. It is a process that consists of generating these two particles and subsequently entangling them together. After separating

them with a distance, they still influence each other. Conducting computations with them is therefore significantly more effective [1].

1.2 Quantum key distribution

Quantum key distribution (QKD) is one of the key elements of quantum cryptography. This process involves two sides negotiating and producing one identical secret key that can be used to encrypt or decrypt messages. The advantage of quantum key distribution is that throughout the negotiation, users can determine whether any third party is trying to listen to this negotiation, thanks to fundamental property described in section 1.1, that being the observation of particles. By introducing a third party to the communication, they bring anomalies, disturbances, and errors to measurement. Prior to negotiation, both parties determine a threshold anomaly rate. When these detected anomalies are at their lowest, the key is declared to be secure, otherwise, the protocol is aborted and would need to be recreated. When the negotiation is finished, the quantum key can be used as a symmetric key for encryption by protocols such as Advanced Encryption Standard (AES) [2].

1.2.1 BB84 protocol

Protocol BB84, named after its inventors Charles H. Bennett and Gilles Brassard, was published in 1984 and primarily focused on photon polarisation as quantum entities. In the modern era, optic fibre communication commonplace. Through a quantum channel, two sides will communicate and exchange information. We commonly refer to these two sides as Alice and Bob, where Eve is a third party wanting to peek into the communication [3].

The protocol is designed with quantum indeterminacy in mind (see section 1.2). those being quantum indeterminacy, which means without measuring the particles you can't determine their current state. However, measuring such particles forces them to collapse into one of their final states.

The protocol starts with a base. This base refers to a polarised pair of entities with a rectilinear basis, commonly referred to as + base, with either 0 or 90–degree rotation, and a diagonal basis, commonly referred to as **X** base, with either 45 or 135–degree rotation. We can describe this communication in 5 steps, see also table 1.2.1:

1. Alice generates a random n bits by a random number generator.
2. Alice randomly generates bases, either + or **X**.
3. Alice polarises photons according to negotiated polarisation table, see table 1.2.1.

4. Bob randomly generates its receiving bases.
5. Bob measures received photons.
6. They both publicly discuss on which bases they matched.
7. Roughly 50 % of measured bits are lost in a process and shared key in established.

Without any knowledge of Alice's base, Bob can just guess any of his bases. When bases do not match, Bob will measure the incorrect states. If Eve was eavesdropping, it would introduce errors to Bob's measurement. This would make the key insecure and it would be discarded. Even without any eavesdropping, the channel can introduce errors to the measurement due to environment properties [3]. This means that communication by long distance is not possible on its own. The network needs to provide so–called repeaters. These devices repeat received photons. But these devices are a significant security problem. If these devices got into hands of an attacker, they could see every single communication going through them. With modern symmetric ciphers needing at least a 256–bit long key to be secure, Alice would need to generate at least 512 random bits [3].

Tab. 1.1: Polarisation table example

Base	0	1
X	↗	↘
+	↑	→

Tab. 1.2: Example of BB84 key negotiation

Alice's random bit generation	0	1	1	0	1
Alice's random base generation	+	+	X	+	X
Alice polarisation	↑	→	↘	↑	↘
Bob's random base generation	+	X	X	X	+
Bob's measurements	↑	↗	↘	↗	→
Secret key	0		1		

1.2.2 Coherent One–Way Protocol

Coherent One–Way Protocol (COW) was published in 2012 and is a key exchange protocol over optical fibre cable. The transmission consists of two parties, generating states of logical 0 and 1 and generating decoy signals to confuse potentially eavesdropping third parties. As the name suggests, it is based on coherent pulses generated by Alice. Every bit of information is encoded into a pair of pulses. This

pair consists of either μ pulse, which represents a small amount of photons being sent, or 0–pulse, which represents no photons being sent. By combining these two states we can modulate a logical 1, shown in formula 1.1, as a pair of pulses consisting of a μ pulse, followed by 0–pulse. For logical 0, shown in formula 1.2, the pair order is reversed, which means modulating as pair 0–pulse followed by μ pulse. Decoy, shown in formula 1.3, is modulated as a pair consisting of two μ pulses in succession [4] [5].

$$\text{Logical}\,1 : |0> + |\mu> \tag{1.1}$$

$$\text{Logical}\,0 : |\mu> + |0> \tag{1.2}$$

$$\text{Decoy} : |\mu> + |\mu> \tag{1.3}$$

On Bob's end, he has a data–line detector D_{m1}, with which he detects incoming pulses. To ensure security, Bob randomly measures coherence between two successive μ pulses by detector D_{m2}. This can happen either by measuring a decoy pulse or two successive non–decoy pulses. Detector D_{m2} will evaluate if wavelengths and phases of received μ pulses are aligned. If they are aligned, we surely know, that no interference was added throughout the transfer. If it detects any misalignment, a loss of coherence is introduced, thus making the key insecure and the key is discarded [4] [5].

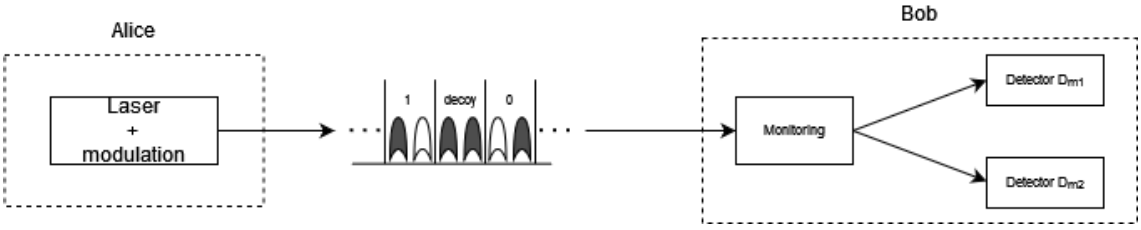

Fig. 1.1: Coherent one–way protocol

The biggest advantage over polarisation protocols is easier a cheaper implementation over optical fibre channels without the need to add any repeaters along with longer distances by making receivers passive optoelectronic components, all whilst still maintaining a high transfer rate.

Experiments attempting a long–running key exchange from Geneva to Neuchatel in Switzerland over the distance of 150 km for 10 hours could achieve a 2.5 kbps real–time bit rate with an average error rate during 3 hours of 2.5 bps with a 43 dB loss line by using superconductor components [6].

2 Post–quantum cryptography

Cryptography is an essential part of everyday modern-day communication, from digital signatures using RSA, to hash functions in the SHA algorithm. A lot of legacy and new hardware uses some sort of cryptographic algorithm. They are based on some sort of mathematical problem and fundamentally restructuring cryptography would be an almost impossible task. The most common mathematical problems are prime number factorisation and discrete logarithm problems. But with the rise of quantum computing, future attacks with Shor's algorithm could theoretically solve these problems in polynomial time, thus putting commonly used cryptographic algorithms at risk. Endangered algorithms are shown in table 2. It is predicted that within the next twenty years, mankind could produce a quantum computer powerful enough to break algorithms like DSA or RSA [7] [8]. This generates a significant opportunity for the relatively young research area of the post–quantum cryptographic systems. Researchers are introducing mathematical operations with little to no computational speed advantage on quantum computers.

Tab. 2.1: Security of algorithms against quantum computers by NIST

Algorithm	Type	Impact of quantum computer
AES	Symmetric key encryption	Larger keys (at least 512 bits)
SHA–2, SHA–3	Hash functions	Bigger outputs required
Elliptic curves	Signatures, key exchange	No longer secure
DSA	Signature, key exchange	No longer secure
RSA	Signature, key exchange	No longer secure

Currently post–quantum cryptography research is separated into 6 different branches [10]:

- Lattice–based cryptography.
- Multivariate cryptography.
- Hash–based cryptography.
- Code–based cryptography.
- Supersingular elliptic curve isogeny cryptography.
- Symmetric key quantum resistance.

2.1 Lattice–based cryptography

Lattice–based cryptography is based on a fundamental principle of a lattice, which is a set of points in n dimensions with a periodic structure [8][9]. We could imagine

a two–dimensional lattice as a sheet of graph paper. Every square is a single point in a finite lattice space.

Definition 2.1.1 *A lattice $L \subset R^n$ is the set of all integer linear combinations of basis vectors $b_1, \ldots, b_n \in R^n$, or $L = \{\sum a_i b_i; a_i \in Z\}$.*

Some of the most popular protocols are Crystals—Kyber, NTRU and SABER. The main mathematical problems that are theorised to be unbreakable by quantum computers are the Short Vector Problem (SVP) and Closest Vector Problem (CVP).

2.1.1 Shortest Vector problem

The shortest Vector problem is an optimisation on a lattice. When given base B, consisting of 2 vectors a_1 and a_2, finding a non–zero vector with their linear combination measuring N. In other words, finding a non–zero vector such that:

$$N(v) = \lambda(L), \tag{2.1}$$

where λ is Euclidian norm of said vector. It is believed that to be NP–hard problem, with no solution so far.

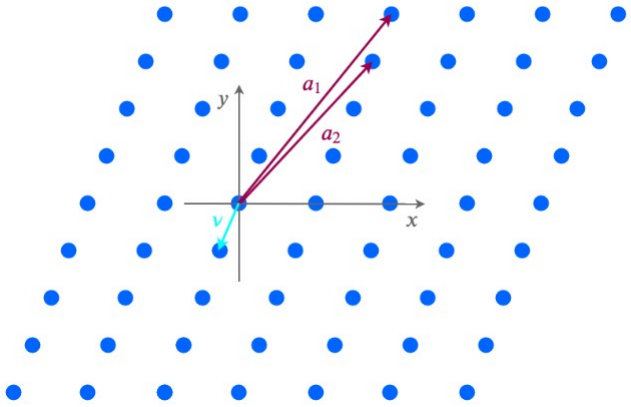

Fig. 2.1: Shortest Vector problem on 2–dimensional lattice

2.1.2 Closest Vector problem

Similar to SVP, the Closest Vector problem is an optimisation problem on a lattice. Given a base B, consisting of two vectors a_1 and a_2, and a target vector t and finding a linear combination of base vectors such as they will be closest to t. It's believed that CVP is an NP–hard problem.

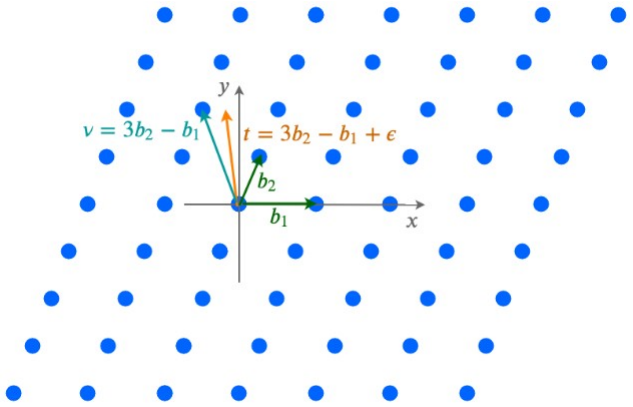

Fig. 2.2: Closest vector problem on 2–dimensional lattice

2.2 Multivariate cryptography

Multivariate cryptography is asymmetric cryptography over multivariate polynomials over a field F. Solving multivariate polynomials is proven to be an NP–complete problem. That is why they are considered a good candidate for post–quantum cryptography, especially for signature schemes because of their shortest signature length [10].

2.3 Hash–based cryptography

Hash—based cryptography is commonly used for digital signature schemes, computational proof of integrity, and more. Merkle's signature scheme, using Merkle's tree is one of the examples of a one-time signature. A one—time signature is a type of signature that can be used to sign only a single message. A combination of more one–time key pairs is an advanced method of using key pairs a finite number of times. By revealing parts of Merkle's tree, it is possible to verify a given hash with its corresponding input. Hash-based signatures rarely rely on used hash functions. Cryptocurrencies are one of the real-world uses of hash-based cryptography [10].

2.4 Code-based cryptography

Code—based cryptography is a cryptography scheme that introduces errors to encrypted text in a manner that only a certain error correction codes could repair, thus decrypting the whole message. The first and maybe the most popular code–based encryption was presented by Robert J. McEliece in 1978. By introducing errors to ciphertext, only the owner with the private key, in this case, a binary Goppa code matrix, could correct those errors introduced by a public key. This scheme remains

unbroken even after more than thirty years of constant research, making it still secure, but not particularly resource efficient. This protocol will be described in more detail in section 3.1 [10].

3 NIST standardisation

NIST (the National Institute for Standards and Technologies), founded in 1901, is a United States government body attempting to evaluate and standardise technological and scientific projects for everyday life. Post–quantum cryptography is one of the many projects that it is currently concerned with. They initiated a competition in December 2016 to support post-quantum cryptography. There are two topics in this competition. The first is Public–key Encryption and Key–establishment Algorithms and the second is Digital Signature Algorithms. These submissions are currently undergoing public review. This thesis will be focused on Round 3 Finalists of Public–key Encryption and Key–establishment Algorithms [11].

3.1 McEliece

As already mentioned in section 2.4, McEliece is a code—based public-key cryptosystem. McEliece is designed to be one-–way, meaning that it is impossible to find plain text just from a ciphertext and a public key. The following process is simplified. McEliece cryptosystem shares common security parameters [12]:

- n - Length of the Goppa code.
- k - Dimension of the Goppa code.
- t - Guaranteed error–correction capability.

3.1.1 Key generation

Alice's first step is to choose a linear code C, which is an error–correction code that she knows an effective algorithm to solve, thus making the code C public. By making the algorithm a secret, she effectively did not share any secret information, because the parameters generating C are required. For example, generating C from a polynomial is near–impossible to replicate, given a long enough polynomial. In the case of McEliece, binary Goppa code is in place of C A large number of official parameter sets have been proposed. Every set has its advantages and disadvantages. Generating is as follows [12]:

- Generating $C = (n \times k)$, which is possible to correct t errors. G is a generator for C.
- Compute $S = (k \times k)$ random binary non–singular matrix, such as it's easy to compute S^{-1}.
- Compute $P = (n \times n)$ random permutation matrix, such as it's easy to compute P^{-1}.
- Compute $\hat{G} = SGP$; $\hat{G} \in Z^{k \times n}$.

Thus public key is defined by pair $(\hat{G}, t)$ and private key by pair (S, P).

3.1.2 Key encapsulation

The only assumption is that m is a length of k. Afterwards Bob does:

- Compute $m' = m\hat{G}$.
- Generating n–bit vector z with t ones (errors).
- Compute $c = c' + z$.

Bob sends c to Alice.

3.1.3 Key decapsulation

After receiving cipher-text c, Alice does the following:

- Compute $\hat{c} = cP^{-1}$.
- Alice uses decoding algorithm f to decode $\hat{m} = f(\hat{c})$.
- Compute $m = \hat{m}S^{-1}$.

One of the biggest limitations is the size of the public key in the scale of megabytes and key generation is compared to other algorithms much slower. To make each public key efficient, one side should use a generated public key long enough to offset the costs of generating another. On the other hand, encryption and decryption on hardware accelerators are proven to be efficient. The biggest advantage of the McEliece cryptosystem is its ciphertext size, just under 256 bytes, which is perfect for network transport, due to the fact that the whole ciphertext can be fit into one packet [12].

3.2 Crystals–Kyber

Protocol Crystals-–Kyber is a protocol from a family of crystals which are cryptographic primitives on lattices. Crystals-–Kyber is focused on key encapsulation mechanisms (KEM), while Crystals-Dilithium focuses on digital signature. Both protocols are finalists in NIST's post-quantum cryptography competition. Crystal–Kyber is based on a Learning–with–errors problem (LWE) with an alteration on using Ring-LWE instead of Module–LWE, as per most encryption schemes. The protocol itself can work in two modes. CCPAKE is a private key encryption. CCAKEM is a CCPAKE extension KEM scheme based on similar functionalities as CCPAKE. Further, only CCAKEM will be described [13].

Firstly, parameters required for Crystals–Kyber are:

- n - bit size of the encapsulated key (recommended 256, could be used smaller but it will introduce lower noise levels, this reducing overall security).

- k - fixed lattice dimensions.
- q - small prime number satisfying $n \mid (q-1)$ to enable fast NTT–based multiplication, there are documented two known primes, where this equation holds, those being 257 and 769, but they are not recommended due to high enough probability failure, thus recommended value is 3329.
- η_1, η_2 - defining noise levels while encrypting.
- d_v, d_u - security parameters.

With parameters described above, failure probability of $< 2^{-139}$ is achieved for KYBER512. Other parameters are described in table 3.2.

Tab. 3.1: Recommended parameters for Crystals–Kyber

Protocol	n	k	q	η_1	η_2	(d_u, d_v)	Failure probability
KYBER512	256	2	3329	3	2	(10,4)	2^{-139}
KYBER768	256	3	329	2	2	(10,4)	2^{-164}
KYBER1024	256	4	329	2	2	(11,5)	2^{-174}

3.2.1 Key generation

Key generation function named CPAPKE.KeyGen() produces a private and secret key, via this process:

- A seed $p = \{0,1\}^n$; $n = 256$ is generated and publicly announced .
- Matrix $\hat{A} = M^{k \times k}$ is generated by seed and Shake–128 hash function in NTT domain.
- Secret coefficient s and and error e are generated by centred binomial distribution (CBD).
- Through NTT we get $\hat{s} = NTT(s)$ and $\hat{e} = NTT(e)$.
- Thus its possible to compute private key $\mathbf{pk} = As + e$ and making secret key $\mathbf{sk} = \hat{s}$.

The result of this operation is a key–pair of private and secret key [13].

3.2.2 Key encapsulation

By using keys from section 3.2.1, we may proceed to key encapsulation, which involves negotiating common secret keys on both sides. **Pk** and **Sk** was well as previously established common parameters. For simplification, let us assume that one side, Alice, wants to establish a common secret key with the other side, Bob [13].

- Alice, through a RNG or PRNG generator generates a 32–bit value m (generator and its initial parameters should be kept in secret).

- Alice uses SHA–3–256 to hash m.
- $(\bar{K}, r) = G(m, H(pk))$, where H is SHA–3–256 and G is SHA–3–512 and r is called random coins.
- Values (pk, m, r) are send to Bob.
- Bob, again, generates $\hat{A} \in R_q^{k \times k}$ in NTT.
- Bob by centred binomial distribution samples r, e_1, e_2.
- Bob compute $u = NTT^{-1}(\hat{A}^t \circ r + e_1)$.
- Bob compute $v = NTT^{-1}(\hat{t}^T \circ r) + e_2 + Decomp(m))$, where $Decomp$ is a decompression function.
- $c = Comp_q(u, d_u), Comp_q(v, d_v)$, where $Comp$ is a compress function.
- Bob encrypts c with Alices pk and sends it to Alice.

3.2.3 Key decapsulation

After encapsulating the key, both sides have their respective ciphertexts c and their secret keys Sk, thus making it possible to decapsulate the ciphertext (shared key) by steps. Both sides do the following:

1. $u = Decomp_q(c, d_v)$,
2. $v = Decomp_q(c + d_u \cdot k \cdot \frac{n}{8}, d_v)$,
3. $m' = Comp_q(v - s^T \cdot u, 1))$,
4. $(\bar{K}', r') = G(m', Sk)$,
5. Both sides tries encapsulation process one more time from section 3.2.2, but without sending the resulting c, naming it c',
6. If $c = c'$, then we can compute the final key $K = \text{KDF}(\bar{K}', H(c))$, where `KDF` represents SHAKE–256 and H represents SHA–3–256,
7. If $c \neq c'$, the process should be repeated from the start due to errors.

In the end the common secret key K is established and available for symmetric encryption [13].

3.3 NTRU

NTRU protocol is another NIST finalist in the key encapsulation mechanism category. Similarly to other finalists, NTRU is a encapsulation method based on lattices and around a Shortest Vector Problem (SVP), described in section 2.1.2. Additionally, a Learning with Errors (LWE) problem is introduced, which brings a small enough error into the equation that can possibly mitigate any possible attacks. By introducing error e, we arbitrarily increase the difficulty of finding a given vector that would be close to point zero, will not be a zero–vector, and will still belong to the lattice [15]. NTRU operates on a special subset of lattice called polynomial rings,

which is a ring-shaped set of polynomials over a lattice . A Gaussian noise is most commonly for error distribution. NTRU is additionally known as a probabilistic algorithm. Security of NTRU comes from mixing polynomial systems [14].

Key generation

NTRU relies on three main integer parameters `N, p, q` and four sets of polynomials of degree $N-1$. P and q does not necessarily have to be prime numbers, but their greatest common divisor (GCD) should be 1. The process of key generation is as followed:

- Alice chooses 2 polynomials f and g, which both lies in the given Lattice ring.
- Additionally, polynomial f has to have inverse in modulo p and modulo q. This principle is shown in formula 3.5.
- This inverse values are denoted as F_q and F_p.
- Alice then computes equation:

$$h \equiv F_q \times g(modq). \tag{3.1}$$

- Alices public key is polynomial h and polynomial f is hers private key.

Despite the fact that NTRU was originally created as Key Encapsulation method (KEM), it can be also used as digital signature algorithm. This variant is also publicly available as open-source code.

Key encapsulation

Let's say that Bob wants to negotiate a shared key with Alice. So the process is simplified:

- Bob randomly generates a shared key m.
- Bob requests Alice for her already generated public key h.
- Bob randomly chooses a polynomial ϕ that lies in the ring.
- Bob computes equation:

$$e \equiv p \times \phi \times h + m(mod\, q). \tag{3.2}$$

Ciphertext e is encrypted with Alice's public key, thus making it unreadable for everyone without an appropriate private key.

Key decapsulation

After Alice receives ciphertext e she is ready to decapsulate this shared key with her private key f:

- Alice computes a as:

$$a \equiv f \times e(mod\, q). \tag{3.3}$$

- Alice chooses coefficients of a in interval $(-q/2, q/2)$.
- Finally Alice can recover message by computing:

$$F_p \times a (mod\, p). \tag{3.4}$$

As the name probabilistic name stands, there is a low possibility that recovering message from given parameters will fail. Thus its recommended to add additional checks to ensure decryption will work.

$$F_q \times f \equiv 1 (mod\, q) \cup F_p \times f \equiv 1 (mod\, p). \tag{3.5}$$

3.4 SABER

SABER protocol is the final NIST round, with three finalists in the standardisation competition. It is based on a lattice and heavily relies on Module learning with rounding problems (Mod-LWR). This problem is based on the Learning with rounding problem (LWR), which introduces a small enough error e to mitigate any possible attacks on a given ciphertext. This problem is a close child to a Learning with error problem (LWE), but with small caveats, such as different distributions of introduced errors e [15] [16].

Firstly, SABER defines recommended parameters for its successful operation:

- p, q, T - Modulo values, chosen to be power of 2, that means $q = 2^{\epsilon_q}, p = 2^{\epsilon_p}, T = 2^{\epsilon_T}$, where $\epsilon_q > \epsilon_p > \epsilon_T$. Higher values will result in lesser security but higher correctness.
- μ - Coefficients of secret vectors s, sampled to the centre of binomial distribution. where $\mu < p$. Higher values will result in lesser security but higher correctness.
- n, l - Defines a polynomial ring, thus defines secret vector s and determines lattice problem $l \times n$. Higher values increases security but reduces its correctness. Recommended values for $n = 256$.
- F, G, H - Hash functions used in protocol. Functions F and H are hash functions SHA3–256, function G is hash function SHA3-512.
- *gen* - Output function for generating pseudo–random matrix A.

Key generation

SABER generates its key as followed:

- Seed is chosen from $\{0, 1\}^n$.
- Random matrix A is generated from seed as:

$$A = gen(seed) \in R_q^{lxl}. \tag{3.6}$$

- Secret vector s is sampled as binomial distribution.
- Value b is calculated from equation 3.7 using matrix A and vector h. Afterwards right bit shift operation is used. This operations replaces rounding, but still resulting into same output.

$$b = ((A_T s + h) mod\, q). \tag{3.7}$$

As a result of this generation private key pk is a pair of values $seed$ and b. Public key pk is value s.

Key encapsulation

As already mentioned, SABER is a probabilistic protocol. That means that communication parties can and sometimes will fail to establish a shared key. But this probability is mostly negated by sending additional control parameters [16]. Lets say that Alice already generated her private and public key. Afterwards, the protocol is as followed:

- Alice sends her public key, those being $seed$ and b_a.
- Bob calculates s_b and matrix A from received Alices seed. Then he calculates values b_b and v_b as followed:

$$b_b = A^T s_b + h. \tag{3.8}$$

$$v_b = b_A^T s_b + h. \tag{3.9}$$

- Bob sends Alice calculated value b_b.

Key decapsulation

After Alice receives value b_b she can proceed to decapsulation as:

- Alice afterwards computes value v_a as:

$$v_a = b_b^T s_a + h. \tag{3.10}$$

- Alice and Bob can now compute shared key k.

If whole communication went without any problems, Alice and Bob will have shared key k. This algorithm is described without correction parameters.

4 Key combination

The main goal of this chapter is to introduce a problem that could affect the future of quantum and post–quantum cryptography. With the rapid development of modern computer technologies, we are never able to accurately predict the future state of cryptography. As discussed in section 1.2 and chapter 3, several future problems have been identified. Both QKD and PQC key distribution methods are vulnerable with the development of quantum computing. Thus, a significant number of precautions and alterations need to be determined.

This problem presents an opportunity to develop an approach that blends both quantum and post-quantum cryptography protocols. By choosing a specific method, we can combine two keys into a one shared hybrid key with these properties. This process ensures that if one protocol is compromised, the other one will still hold, thus securing the entire communication.

4.1 Xor based combination

Logical operation XOR is one of the fundamental components of cryptography. It is a binary operation over any given two inputs, which produces one output, see table 4.1. It is based on a logical gate, thus making it extremely easy to hardware accelerate. The usual bottleneck for this function is that it is a stream cipher. Steam ciphers input their source bit by bit. This behaviour is better suited for real–time applications, where inputs are procedurally generated. On the other hand, block ciphers compute over a block of information. This may create a bottleneck in generating inputs [17].

Input A	Input B	Result
0	0	0
0	1	1
1	0	1
1	1	0

4 mathematical properties make XOR function very useful in modern computer science:

- Commutative - mean that it does not matter the order of inputs, thus $A \circ B = B \circ A$.
- Associative - chaining together more XOR operations is possible and it does not matter in which order inputs are fed, for example $A \circ (B \circ C) = B \circ (A \circ C)$.
- Identity element - Any input combined with 0 stays unchanged, $A \circ 0 = A$.

- Self–inverse - Any input XOR'd with itself will result in 0, $A \circ A = 0$.

XOR is often referred to as a perfect cipher or unbreakable cipher. If inputs are truly random, the result will have enough entropy to withstand any cryptography attack. If input A is a plain–text and input B is a key, then B should be generated from a truly random generator to ensure perfect balance between zeros and ones in the result. Otherwise, the XOR cipher is susceptible to a frequency analysis attack. In case of two randomly generated keys, we can't be certain if keys have sufficient entropy to negate this attack.

Although the XOR function may appear to be the ideal candidate for a key combination of this type, the opposite is in fact true. As already mentioned, the commutative property of a XOR function is a significant benefit, but also a drawback. Two inputs, key A and B, are combined into key C, it is possible to compute the remaining unknown part if the two keys are leaked. This means that XOR functions are reversible. Even in this edge case, it is a significant flaw [17].

With these properties in mind, being reversibility and vulnerability to cryptoanalysis, XOR is not suitable for key combination. In theory, it is a suitable candidate if key generation could be purely random, if keys are destroyed after combination, or if keys are securely stored. In reality, none of these parameters can be met with certainty. Fundamentally, using this function contradicts the basic principles of this thesis.

4.2 Hash based combination

A hash function is a one—way function that takes an arbitrary sized input and produces an output of a fixed size. Ideally, this process should be irreversible, and a specific input should only produce one given output. But so—called collisions are bound to occur. With hash functions we have these types of collisions:

- Collision– Resistance - collision occurs when hashing two different inputs results in the same hash value, mathematically said that $H(M) \neq H(M'); M \neq M'$, the complexity of finding this collision is $2^{\frac{n}{2}}$, where n is a length of the hash output.
- Pre–image Resistance (Pre) - given hash function H and hash output $H(M)$, it should be computationally infeasible to retrieve the original message M or generate any other M that would generate the same output, with a complexity of 2^n, where n is a length of the hash output.
- 2nd Pre–image resistance (Sec) - given hash function H and message M, it should be computationally infeasible to find a different message M such as it would produce the same hash output, making this the weakest and the most attacked point of most hash functions due to a "brute–force attack".

All hash functions should have three security properties. Identifying so–called collisions does not mean that a hash function is deemed to be broken. However, proving that there is a way to exploit a given hash function will mark the hash function as less secure. Finding these weak spots, so–called structural weaknesses, may move theoretical attacks to practical ones. A prime example is the hash function MD5, which was invented in 1991, and was officially broken in 2005, due to a so–called birthday paradox. Simply said, MD5 produces a 128–bit long hash that is too small, thus making collisions occur much more frequently. Yet MD5 is still widely used despite the fact that it has been broken. For example, a file fingerprint called md5sum is a simple check, if the file was maliciously changed. This function takes the whole file and reduces it into a hash string. This string can be verified by user by simply hashing whole downloaded file by himself and then comparing those two hashes. It is still used since its computationally simple and fast, thus hashing big files is less demanding [18].

Hash functions are commonly used for a variety of applications, such as a file or message integrity verification. In each message or file, a hash of a that attachment is given, making it easy to verify if it was tempered with it. Similarly, the digital signature also uses some form of a hash function to ensure that a given message came from a given user and that it was not tampered with by adding a unique identifier. Another use of hash functions is password storage when passwords are not kept in a server's database as plain–text but are rather hashed. Even in the case of a security breach, attackers should not be able to retrieve original passwords [18].

A commonly used hash functions is a family of hash algorithms known as SHA (Secure Hash Algorithm). SHA–1 is a variant of a hash function which is declared to be insecure, but not yet broken. SHA–2 hash algorithms are commonly used with its variants like SHA–2–128 and SHA–2–256 and not thought to be quantum-secure. This makes SHA–2–512 a better candidate for a quantum–resistant hash function. The last member of the SHA family is SHA–3, also known as SHAKE, which was released in 2015 by NIST. As opposed to SHA–2, the name of a SHAKE function, like SHAKE–128 or SHAKE–256, does not imply its output size like in the case of SHA–2, but rather its security strength. It is thought that SHA–3 should be a quantum–resistant hash function [18]. Differences between SHA–2 and SHA–3 family, see table 4.2 are:

- Resistance to length extension attack: SHA–2 is more vulnerable to these types of attacks because if a message authentication code is known for SHA–2, it is easier to forge an authenticator on other hashed. This the main reason for developing an HMAC function. SHA–3 solved this problem by key–prefix construction.
- Performance: SHA–2 has higher performance compared to SHA–3, shown in

table 4.2.

- Different internal design: SHA–2 is designed around Davies–Meyer structure with block ciphers, built on ARX network. SHA–3 is designed on sponge structure with Keccak permutation [26] [27].

Tab. 4.1: SHA algorithms, their release year, collision resistance, bit sized output and design architecture

Hash	Year	Collision resistance	Bit size	Design
SHA–0	1993	80	160	32–bit ARX DM
SHA–1	1995	80	160	32–bit ARX DM
SHA–2–256	2002	128	256	32–bit ARX DM
SHA–2–512	2002	256	512	64–bit ARX DM
SHA–2–512/256	2012	128	256	64–bit ARX DM
SHA–3-256	2013	112	224	64–bit Keccak sponge
SHA–3-512	2013	256	512	64–bit Keccak sponge
SHAKE–256	2013	<256	any	64–bit Keccak sponge

In theory, hash functions could be the solution for key combinations. Its fast implementation should not cause any bottlenecks. This method should be secure if a strong enough hash function is used, that being SHAKE–256, SHA–3–256, or SHA–2–256 for an output of 256 bits or its alternative counterparts for 512–bit outputs. However, as part of this process, we lose possible flexibility with the size of outputs. Currently, ciphers like AES use 128, 192 or 256–bit keys. That presents an obstacle to implementing other variants. If we would like to use AES–128, it is very insecure to split a 256–bit key into two separate keys. That creates a vulnerability to attack, because by splitting the key, collision resistance drops by half of the original value. A similar situation will occur if we would like to use a 192–bit key. Reducing the output hash would weaken the hash function itself. Therefore, we are forced to use only 256 or 512–bit keys. Hash functions are a middle ground between security and computational efficiency and a good choice for modern computers and on less–computationally fast devices.

4.3 Key derivation function

Key derivation function (KDF) is a cryptographic function that derives one or more inputs, such as a password or secret key, into a more desirable single or multiple output. It is commonly used for obtaining keys of a desired format, length, or volume, where the original key does not have these properties.

Key derivation functions have a lot of useful applications in cryptography, such as transforming passwords or passphrases into a stronger secret key. This is commonly used in Wi–Fi Protected Access 2 (WPA2) where multiple secret keys are derived from a secret password. These derivations are commonly part of a larger protocol for key agreement such as standard IEEE P1363 [19], or in Key stretching and key strengthening.

Key stretching and key strengthening is a method of deriving keys from a secret passphrase or password that makes an attack extremely slow due to the fact that they are made to be deliberately slow. This dissuades brute-force attacks. One of the possible methods of key strengthening is to add a few parameters to the process. Those parameters are salt and number of iterations [21].

Salt

Salt is a random value which can, but does not have to, be public, and that serves as an extra layer of protection against pre–calculated tables of results of a given algorithm. For example, hashing the same input will always result in the same output. Therefore, adding salt to the password will negate, or at least slow down, incoming attacks and possible cracking, because two users could theoretically have the same password, resulting in the same hash value. The only requirement of this method is that the salt value has to be stored with the hash value of the password in the database.

Hashed passwords will be recalculated when a user tries to login into the service. Salt should not be reused and ideally should not be too short. This puts a designer of a given system into a position where they have to choose between greater security (larger salt length) and a greater resource demand on data storage, or weaker security (smaller salt length) and a smaller data storage requirement [22].

4.3.1 Extract–and–Expand Key Derivation Function (HKDF)

Extract–and–Expand Key Derivation Function (HKDF) is a candidate for this particular case of a key combination. Published in 2010 as a product of IETF (Internet Engineering Task Force), it has since been publicly approved and is in common usage. It is based on HMAC (Hash–based message authentication code), which is an algorithm that takes secret keys and messages and combines them into one output of a specific length [23].

HKDF follows the extract–and–expand paradigm, where it consists of two modules, extract and expand.

Extract

The first step is to extract a pseudo–random key from the given inputs. By doing the following operation:

$$PRK = H(\text{IKM}, salt, L), \tag{4.1}$$

where output PRK is a pseudo–random key, H is an HMAC hash function, IKM are keying materials (in this case two keys), salt is an optional parameter (when not given, salt is automatically value 0) and L is a length of output PRK. The only concern in this step is the fundamental process of extracting the pseudo–random key out of given information. Therefore, the input information should have well–dispersed entropy on given keys, which will be reflected in the entropy of the resulting key. Another note to this process is the optional use of salt. Salts and their meanings are described in section 4.3. In a real–world scenario, salts are necessary for the protection of outputs against a lot of different attacks. Salts should be a randomly generated value and should never be reused. However, it is not always possible to generate a salt value, so the salt value is set to optional.

Expand

The second stage is the most important part of this algorithm due to the fact that the first is not mandatory. Many NIST and IEEE standards only take into consideration the second part [20]. The computation is as follows:

$$OKM = HKDF(PRK, \text{info}, L), \tag{4.2}$$

where OKM is output keying material, $info$ is similar to salt value, commonly used for binding given algorithm to an application. This value can be the ID of the application, protocol number or any other identifiers. This results in more diversity when using the same inputs. Value L is a number of octets in resulting OKM. OKM is calculated as a result of multiple chained HMAC hash functions together in a way:

$$\begin{aligned} T(0) &= \text{Emptystring} \\ T(1) &= H(PRK, T(0), \text{info}) \\ T(2) &= H(PRK, T(1), \text{info}) \\ T(3) &= H(PRK, T(2), \text{info}) \end{aligned} \tag{4.3}$$

After this computation, the resulting OKM is a concatenated result of T_i, where i is a number of requested octets [23].

In conclusion, KDF is the most secure way to combine two keys. Its effectiveness is demonstrated by its wide usage in protocols like WPA2. However, to achieve the

most security, it is mandatory to use salt. This introduces the problem of generating the same salt value on both ends of the communication to achieve the same key output. One possible solution is a derivation from some shared values, possibly from the keys themselves, or through additional communication. Unfortunately, both approaches come with their own time–consuming operations and weak points. Another advantage is its flexible output size. Compared to hash-based combination, this does not limit individuals to using specific symmetric ciphers due to the static length of hash outputs. Yet key derivations come with a significant computational cost, resulting in problems if used on less computationally powerful devices.

4.3.2 KECCAK Message Authentication Code (KMAC)

KMAC is another widely used pseudo-random function (PRF) and keyed hash function with a variable output based on KECCAK, in other words, based on SHA–3, see section 4.2. KMAC is described in NIST standard SP 800–185 [24]. KMAC exists in two variants, being KMAC–128 and KMAC–256. Both variants are based on function cShake–128 and cShake–256 respectively. KMAC–128 offers 128 bit security against pre—image or second–pre—image attacks, see section 4.2. The biggest drawback and difference between these two variants is speed, while collision resistance is sufficient with both. For the purpose of key combination, only the PRF part of KMAC will be used. NIST refers to the needed parameters as followed:

- X - main input message, can be any size.
- L - length of output hash (in bits), in this case 256.

Computation of hybrid key would look like followed:

$$K_{hybrid} = \text{KMAC256}((K_{PQC}||K_{QKD}), 256), \tag{4.4}$$

where K_{PQC} is already established post-quantum key and K_{QKD} is already established quantum key. Value 256 represents the length of output hash in bits. Both keys are chained together.

In the case of KMAC, it's recommended to use salt, or in this case, a value called the nonce. The nonce value should be generated by a pseudo-random generator with sufficient entropy. This value will afterwards be added to the keys as an input to KMAC. Thus, the resulting equation would look as followed:

$$K_{hybrid} = \text{KMAC256}((K_{PQC}||K_{QKD}||N), 256). \tag{4.5}$$

5 Implementation

The implementation and practical comparison of all combination methods from chapter 4 will be discussed in this chapter. The main goal is to compare and choose the best combination in security, computation speed, and overall performance.

5.1 Testing environment and topology

Testing topology is straightforward without any unnecessary additions or parties. Two clients, Alice and Bob, are hosted as virtual machines in VMWare virtual environment on the operational system Ubuntu. All parties are situated in a laboratory environment at Brno University of Technology. QKD server is a physical machine from company IDQ, more in section 5.1. Keys are downloaded from Application Programming Interface (API) and used afterwards for the hybrid key. Keys are not stored anywhere, thus minimising any risk of reusing the same keys. Alice and Bob are not communicating with the same entity but rather with their corresponding partnered QKD server. QKD entities are connected with quantum channels and the whole quantum key establishment process is done by them, visualised in figure 5.1. Communication with QKD parties requires a certificates, which are saved on said devices.

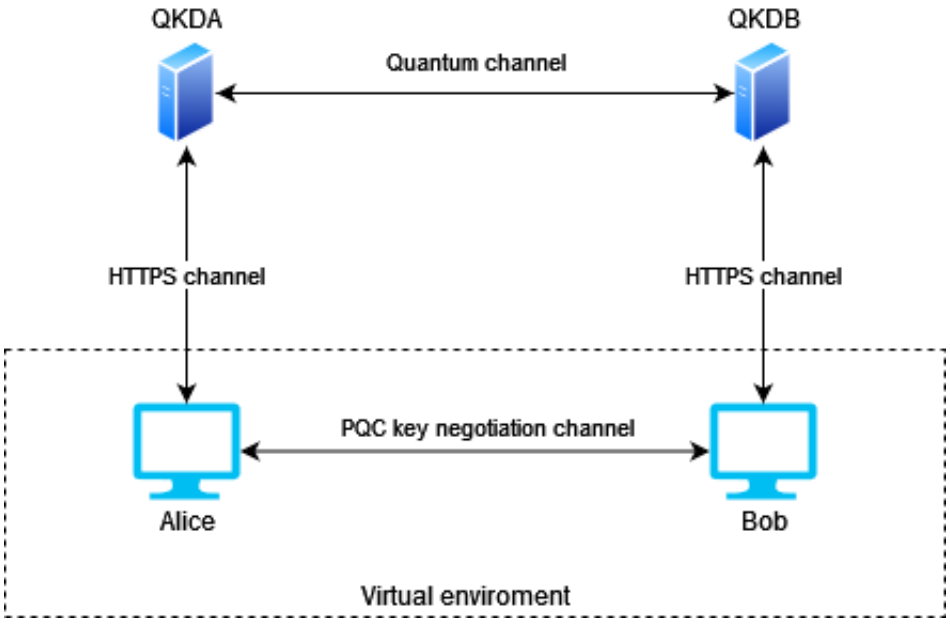

Fig. 5.1: Testing topology

QKD server

As already mentioned, the testing workplace is situated in Brno University of Technology (BUT), specifically in the Faculty of Electrical Engineering and Communication (FEKT) the in Department of Telecommunications, in room T12/SC 5.37, also so called Cryptolab. Two main units, QKDA and QKDB, are connected through an optical quantum channel with FC/APC connectors (see photo 5.2). To ensure the correct functioning of QKD servers, it is necessary to avoid connecting them directly. Additional attenuation devices are used to ensure at least 10dB attenuation on channel. To simulate distance between these two servers, a variable optical fibre cable is used. This cable can be extended up to 25 km. With this scenario, it is possible to achieve attenuation of 0.2 dB/km. Additionally, there is a possibility of introducing a malicious third party into this communication to simulate eavesdropping. Configuration for these servers is done by web interface with tool KEMS, for example adding accepted TLS certificates, restarting, or maintenance.

Fig. 5.2: Photo of QKD servers

The quantum key establishment devices are physical devices from the Swiss manufacturer IDQ. The most essential elements for demonstration purposes are the devices shown in the table below. The devices are connected to using the SSH

protocol, so certificates for each device are also required as well as keys and a certification authority (CA) certificate. As already mentioned, both clients have their

Name	IP address	Certificate	Certificate pass
Alice (QKDA)	192.168.10.102	ENCA-cert.pem	ENCA-key.pem
Bob (QKDB)	192.168.10.107	ENCB-cert.pem	ENCB-key.pem

corresponding partners for communication. QKDA is called master, thus initiating quantum key negotiation, while QKDB is called slave and only answering. This results in QKDB being structurally and visually different then QKDA. Both keys are stored in said devices for a few seconds before deleting them. Keys are paired with their corresponding key ID. With this ID, a key can be downloaded from QKDB.

5.2 Development

Development was run and tested in a Visual Studio Code. It was written in Python 3.9, which is the latest version with the help of a bash support script for downloading keys from a QKD server. Both languages are natively in Linux-based operational systems, thus no additional installations are required. User manual is created for easy operation of demonstrator in attachment A.

Control arguments

For console application control arguments are essential to change dynamic parameters with every run. Thus script required these arguments:

- --initComm or --listener - defining if client will initiate communication or rather wait with open port for initiation respectively.
- --dest [ADDR] - defines destination IPv4 address (without port).
- --local [ADDR] - defines local IPv4 address of exiting interface (any client can have multiple exit interfaces, thus this functionality).
- --mPath [PATH] - defines path to file that initialising client want to send (only initialising client takes advantage of this parameter).
- --sPath [PATH] - defines save path for received file for listening client (only listening client takes advantage of this parameter).
- --comb [xor/sha2/sha3/kmac] - declares mechanism for creating hybrid key, either by XOR-based combination, Hash-based combination (SHA–2–256 or SHA–3–256) or key derivation by KMAC, respectively.

An example of argument parsing is depicted in listing 5.1. By default, port 15000 is chosen for communication. Before this communication, Bob and Alice exchanged their public keys via a secured shared medium to ensure they were not tampered

with. This medium could be a USB stick or a CD drive. Output of said script is depicted in figure 5.3. In this example, the initialising client (Alice) is on right side while the listening client (Bob) is on the left side.

Listing 5.1: Argument parsing

```
1  def main():
2      """
3      Argument parsing
4      """
5
6      parser = argP.ArgumentParser(description='Arguments')
7      parser.add_argument('--initComm', action='store_true')
8      parser.add_argument('--listener', action='store_true')
9      parser.add_argument('--dest')
10     parser.add_argument('--local')
11     parser.add_argument('--mPath')
12     parser.add_argument('--sPath')
13     parser.add_argument('--comb')
14     args = parser.parse_args()
15     config = vars(args)
```

Used libraries

Starting from the top of the script `client.py`, which is the main run file, it is clear that the script uses a library `socket` to make network communication as flexible as possible. With this library clients can freely open and close dynamic ports, thus making peer-to-peer communication easier and clearer without any overheads.

PQCrypto

For post-quantum cryptography a library called *PQCrypto* is used. It is an open-source library available from Github and it is one of the recommended libraries to use for NIST Round 3 finalists. It supports multiple key encapsulations methods, such as Crystals-Kyber, McEliece, NTRU, SABER and many more. All of the variants of these algorithms are also present with their recommended variables mentioned in chapter 3. Unfortunately, the library is not that easily installed, thus reading the official installation guide is highly recommended.

In listing 5.2, a module `pqcrypto.kem.kyber512` is used, which is a Crystals-Kyber 512, however any available module can be used from GITHUB repository [25]. Fetching methods for key generation, encryption, and decryption in straight forward. Alice will generate her key pair by calling function `gen_keys()`. Derivation and

encryption of a secret key are done by method `encrypt()`, where one argument is Alice's public key. When Alice receives ciphertext, she can easily use method `decrypt()` to decipher the shared secret key.

Listing 5.2: Example of generating, encrypting and decrypting derivated key

```
1  from pqcrypto.kem.kyber512
2  import gen_keys, encrypt, decrypt
3
4  # Alice generates a key pair of public and secret key
5  public_key, secret_key = gen_keys()
6
7  # Bob derives a secret key and
8  # encrypts it with Alice's public key
9  ciphertext, plaintext_original = encrypt(public_key)
10
11 # Alice decrypts Bob's ciphertext
12 # to derive the now shared secret
13 plaintext_recovered = decrypt(secret_key, ciphertext)
```

Table 5.2 shows some of the available algorithms and the time needed for generating key pairs, derivation and encryption of keys, and decrypting ciphertexts. It is mainly focused on the Crystals-Kyber family of algorithms. `McEliece 348864`'s key generation time could be discouraging at first, but it must be noted that generating key pairs is not always mandatory for every communication. Key pairs could be reused. In this case, public keys will be pre-shared, thus skipping the first step. In table 5.2, key lengths for a particular algorithm are shown as well as the length of ciphertext on a negotiated secret key.

Algorithms (10 000 runs)	Key generation [ms]	Encrypting [ms]	Decrypting [ms]	Total time [ms]
Crystals-Kyber 512	0.031682994	0.01821944	0.005159678	5.51E-02
Crystals-Kyber 768	0.034731898	0.022210953	0.008181872	6.51E-02
Crystals-Kyber 1024	0.038353925	0.026300926	0.012050421	7.67E-02
McEliece 348864	11.88667321	0.045699596	0.021952629	1.20E+01

Tab. 5.1: Algorithms from library PQCrypto and their time needed

Size (bytes)	Public key	Private key	Ciphertext	Secret key
Crystals-Kyber 512	800	1632	736	32
Crystals-Kyber 768	1184	2400	1088	32
Crystals-Kyber 1024	1568	3168	1568	32
McEliece 348864	261120	6452	128	32

Tab. 5.2: Algorithms from library PQCrypto and their key sizes

HTTPS interface

Requests for QKD servers API are handled separately in a bash script running command `curl`. `Curl` is a tool for transferring data through interfaces with HTTP or a secured variant, HTTPS. For HTTPS, it is required to specify the certificate and its additional necessities, such as the key to the certificate or a certificate from the Certifications Authority (CA). In the example listing 5.3, Alice executes curl command from a Python environment in background shell. She receives the key and its corresponding key ID from her QKD server (QKDA). It is important to note that this pair is stored in QKD machines for only a few seconds before being erased. This ensured that clients will not have infinite time and any malicious party could steal this pair. The key pair is encrypted throughout the communication with attached certificates. Returned QKD keys are in `json` format `['key_ID': 'string', 'key': 'string']`.

Listing 5.3: Alice requesting key from QKD server

```
1 def RestApiRequestGetKey():
2     curl_cmd = f'curl --cert {master.master_cert}
3       --key {master.master_key} --cacert {master.ca_cert}
4             -k https://{master.addr_master}{path_to_key}'
5     output = subprocess.check_output(curl_cmd,shell=True)
6
7     json_data = json.loads(output)['keys']
8     # [{'key_ID': 'string', 'key': 'string'}]
9     return json_data
```

For Bob's side, he only receives the key ID and has to download it from its corresponding QKD server (QKDB). This key is then fed into the already mentioned support bash script listed in 5.5. For correct functionality, it is required that Linux operating system got installed packages `curl` for downloading and `jq` for parsing answers in `json` format.

Listing 5.4: Bob requesting key from QKD server by key ID

```
def RestApiRequestGetKeyFromId(key_ID):
    curl_cmd = f"sh QKDscript {key_ID}"
    try:
        output = subprocess.check_output(curl_cmd,
                                         shell=True)
    except subprocess.CalledProcessError as e:
        output = e.output

    if "null".encode() not in output:
        return output
    else: raise Exception("No QKD key downloaded!")
```

Listing 5.5: Bash script for requesting key from QKDB

```
KMSS_IP='192.168.10.102:443'
Rep=$(curl --cert ENCB-cert.pem --key ENCB-key.pem
    --cacert ca-cert.pem -X POST
        -H 'Content-Type:application/json'
            -d "{\"key_IDs\":[{\"key_ID\":\"$KeyID\"}]}"
            -k https://$KMSS_IP/api/v1/keys/ENCB/dec_keys)
Key=$(echo $Rep | jq '.keys[0].key' | cut -d '"' -f 2)
echo $Key
```

Communication

From the outset, communication between clients is not encrypted. But this is true only for the first two "Hello" packets. Throughout key negotiation, either for quantum or post-quantum key negotiation, communication is encrypted with a corresponding scheme. This process does not reveal any private or important information that could be used maliciously.

Alice will send a Hello message, which represents the initiation for establishing a hybrid key. Bob responds with the same message. After that, Bob loads Alice's public key, which they exchanged before this communication via a USB stick. From that key, Bob generates a shared key. He encrypts it with Alice's public key and sends her the ciphertext. Alice will decipher the message with her private key and reveal the shared key. After this, both sides will establish a post-quantum key.

Establishing the quantum key starts with Alice sending an HTTPS GET request to the QKDA server (master). With the quantum protocol, QKDA and QKDB then negotiate a shared key. QKDA replies with a response in `json` format with the key and its corresponding key ID. This communication is encrypted. Certificates

for these serves should be kept securely stored. Afterwards, Alice sends Bob the corresponding key ID. Bob receives this message and creates his HTTPS POST request, for example in listing 5.5, with a key he received to the QKDB server (slave). If time frames were met, QKDB will reply with a quantum key.

At this point, both clients (Alice and Bob) received the same post-quantum and quantum key. Both can start combining this key, resulting in a shared secure hybrid key. After combining with the defined mechanism, the output hybrid key is set to a length of 256 bits for every method. The hybrid key is used for encrypting with algorithm AES-256 in mode GCM, sending an encrypted file through non-secure public channel. This file is sent via a TCP connection. The other client decrypts this file with the same hybrid key and saves it into its internal storage. If every process was executed properly, both clients will have the same hybrid key and shared file. Output is depicted in picture 5.4, also in manual A.

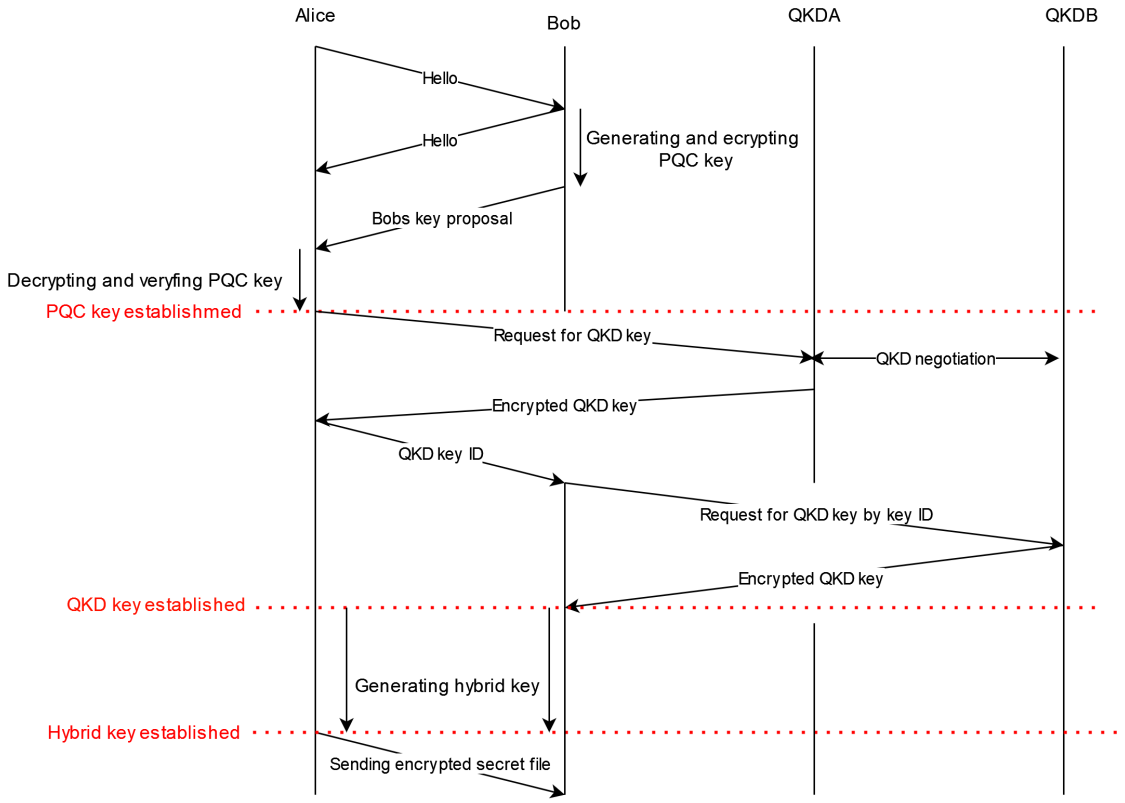

Fig. 5.3: Example of a communication

The whole communication will usually be done in a few milliseconds, which could theoretically increase with the distance between clients or any other interference on the network. However, this should not cause any significant problems because the only time dependent units are QKD servers, which store keys for a few seconds.

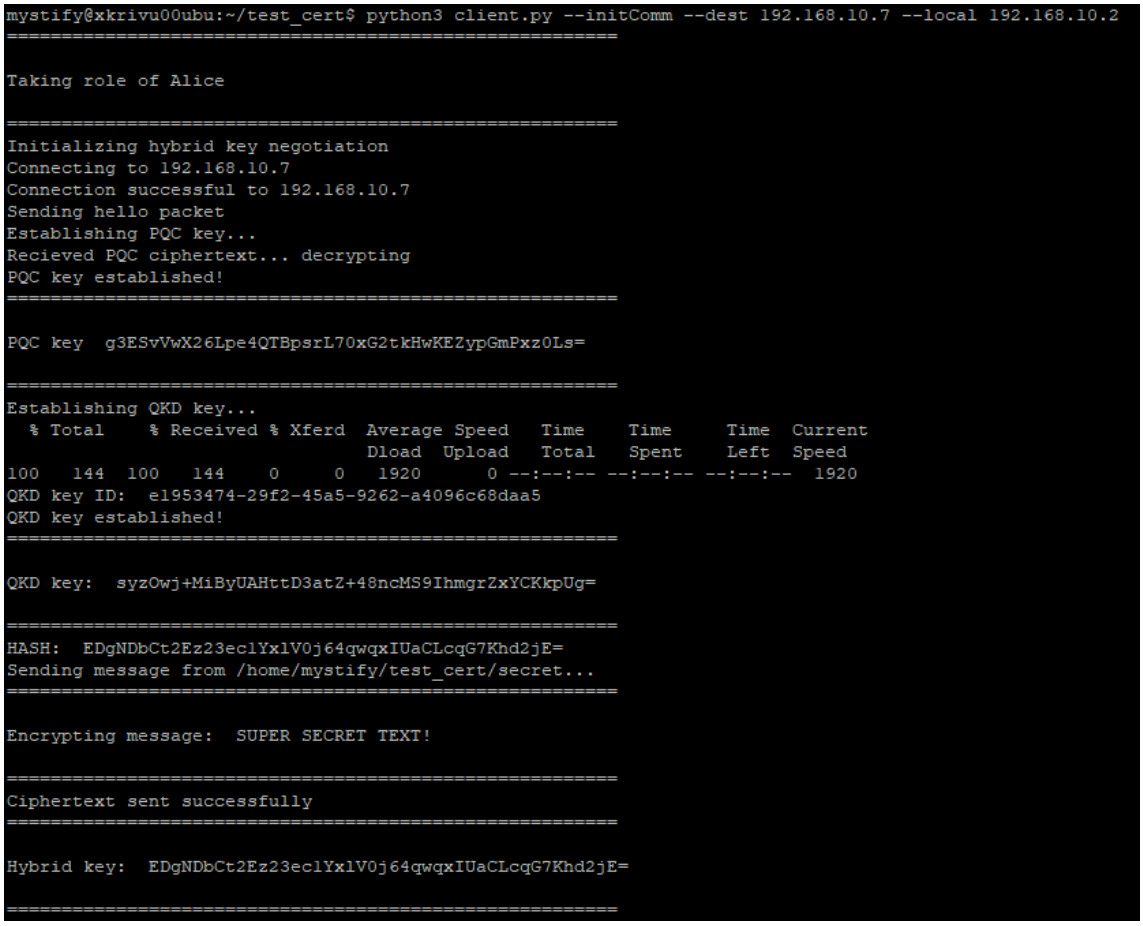

Fig. 5.4: Output of script for Alice

Listing 5.6: Code for initiating communication

```
def initCommThread(ipAddr):
    try:
        t = time.time()
        print("Initializing comm for shared key")
        serverSocketRec = socket(socket.AF_INET,
                                 socket.SOCK_STREAM)
        print(f"Connecting to {ipAddr}")
        serverSocketRec.connect((ipAddr, 15000))
        print(f"Connection successful to {ipAddr}.")
        print("Sending hello packet")
        serverSocketRec.send(b"hello")
        answer = serverSocketRec.recv(4096)
        answer = answer.decode()
        if answer == "hello":
            print(f"Seding hello message to {ipAddr}.")
            #print(public_key)
            #print(f"Public key: {public_key}".encode())
            KEM_encrypted = serverSocketRec.recv(4096)
            print("Recv ciphertext... decrypting")
            KEM = decrypt(secret_key, b64decode(
                                    KEM_encrypted))
            print("Shared key ",b64encode(KEM))
            serverSocketRec.close()
            elapsed = time.time() - t
            print("TIME ELAPSED: ", elapsed)
            return b64encode(KEM)
    except Exception as e:
        print(e)
```

5.2.1 XOR-based combination

XOR, as a basic logical operator is simple to implement. With symbol ^, which represents operator XOR, we can XOR two characters. With two strings it's necessary to create a cycle that will go character by character. Example code is depicted in listing 5.7. This process got a complexity of $O(n)$. In other words, linear complexity. But for reasons already mentioned in section 4.1, this is not a viable option for creating the hybrid key. The output of this combination is set to a length of 256 bits.

Listing 5.7: Code for initiating communication

```
def XorDigest(QKD,KEM):
    return ''.join([hex(ord(QKD[i%len(QKD)]) ^
            ord(KEM[i%(len(KEM))]))[2:]
                for i in range(max(len(QKD), len(KEM)))])
```

5.2.2 Hash-based combination

As mentioned in section 4.2, hash functions will take any arbitrary sized values and convert them into fixed-sized output. However, using any hash functions would be reckless. Quantum safe hash functions should be used, those being SHA-256, SHA-512 and possibly a whole family of SHA-3 should be quantum-safe [27] [26].

Depending on further usage of a hashed key, the size of the output needs to be ascertained. For AES, 256 or 512-bit output is recommended, or whenever the slower SHA-3 algorithms are more important than lesser secure SHA-2. It does not mean that SHA-2 is instantly insecure, it just means that SHA-2 is a more likely to be weakened or broken. Ultimately both families stay secure for the time being [26] [27].

Python library `hashlib` offers widely used hash functions with all of their variants. By simply calling `import hashlib` we have access to many hash functions. Then it's just a matter of choosing preferred hash functions, feeding given input into is and by calling `hexdigest()` returns a hash in hexadecimal format. An example of usage is shown in listing 5.8.

As shown in table 5.2.2, SHA–2–256 performed better then SHA-3-256. This is due to the internal structure of said hash functions. As already mentioned in section 4.2, this provides a secure way to combine these two keys into a hybrid key.

Listing 5.8: Example of usage of SHA hashing functions

```
def Sha2Digest(QKD,KEM):
    s = hashlib.sha256()
    s.update(QKD)
    s.update(KEM)
    sharedKey = s.digest()
    print("HASH: ", bytes.hex(sharedKey))
    return sharedKey
```

By choosing either SHA–2–256 or SHA–3–256, output length is set to 256 bits for further usage.

Tab. 5.3: Time required for each hash algorithm according to given input (10 000 runs)

Algorithm	512 bit input [ms]	256 bit input [ms]
SHA–3–256	0.1233	0.12118
SHA–3–512	0.13636	0.126766
SHA–2–256	0.0756	0.0605
SHA–2–512	0.08116	0.01109

5.2.3 Key derivation

KMAC is chosen to be the implemented key derivation method. As already mentioned in section 4.3.2, KMAC is based on SHA–3. This algorithm is already implemented in library `PyCryptodome` as `KMAC128` or `KMAC256`. For this very use case as PRF, it is recommended to use 256–bit variant. This implementation is very similar to the Hash–based combination from section 5.2.2. This is because the authors are trying to make unified uses of similarly functioning algorithms. KMAC outputs a hybrid key of length 32 bytes (256 bits). Example code is depicted in listing 5.9.

Listing 5.9: Example of KMAC usage

```
1 def KmacDigest(QKD, KEM):
2     mac = KMAC256.new(key=QKD+KEM,mac_len=32)
3     mac.update(QKD)
4     mac.update(KEM)
5     sharedKey = mac.digest()
6     print("HASH: ", bytes.hex(sharedKey))
7     return sharedKey
```

Tab. 5.4: Time required for KMAC to create an output (10 000 runs)

Algorithm	Time required [ms]	Output length [bits]
KMAC128	0.30425	256
KMAC256	0.28251	256

5.2.4 Key utilisation

Negotiating and combining both keys results in one shared hybrid key of length 256 bits. This size is not random, but rather intentionally chosen. Advanced Encryption Standard (AES) is a symmetric block cipher introduced by NIST in FIPS PUB 197 standard. AES supports 128, 192 or 256–bit keys. However, the most commonly

used variant is AES–256 which requires 256–bit key. Thus all already mentioned key combinations result in 256–bit keys.

Furthermore, to increase the security of block ciphers, cryptography created modes for said ciphers. Block ciphers are best suited for 1 block of data to encrypt or decrypt. This means that every further block is encrypted separately without any link to the previous one, creating some link between consecutive blocks. For this usage AES-256 in mode Galois/Counter (GCM) is used because it is widely adopted for its performance.

Library `PyCryptodome`[1] offers variety of different AES modes, among which is mode GCM. The library supports adding nonce to encryption to increase security. This nonce is generated internally in the library. After encryption, the method returns ciphertext and generates nonce and tag, which is a Message Authentication tag (MAC tag). This tag will be used in decryption to ensure that message was decrypted successfully. These values should and will be passed to Bob. However, these values are in the form of non-printable binary characters. Therefore, for transferring, they are encoded into a more readable version, Base64 format. When Bob receives this message, he decodes it back into binary form, decrypts the message, and verifies its correctness. Example of encryption is in listing 5.10 and decryption in listing 5.11.

[1] https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html

Listing 5.10: Example of encryption with AES

```
def encryptAES(Aes_private_key, plain_text):

    cipher_config = AES.new(Aes_private_key, AES.MODE_GCM)

    cipher_text, tag = cipher_config.
                    encrypt_and_digest(plain_text.encode())

    return {
        'cipher_text': b64encode(cipher_text).decode(),
        'nonce': b64encode(cipher_config.nonce).decode(),
        'tag': b64encode(tag).decode()
    }
```

Listing 5.11: Example of decryption with AES

```
def decryptAES(enc_dict, combinedKey):

    regexForPrivateKey = r"(?:Cipher text: )
                            ([^\r\n]+)(?:; Tag: )([^\r\n]+)
                                (?:; Nonce: )([^\r\n]+)"
    regSplitter = re.split(regexForPrivateKey, enc_dict)
    dict_keys = {
        'cipher_text': b64decode(regSplitter[1]),
        'nonce': b64decode(regSplitter[3]),
        'tag': b64decode(regSplitter[2])
    }

    cipher_text = dict_keys['cipher_text']
    nonce = dict_keys['nonce']
    tag = dict_keys['tag']
    Aes_private_key = combinedKey

    cipher = AES.new(Aes_private_key, AES.MODE_GCM,
                                            nonce=nonce)
    return cipher.decrypt_and_verify(cipher_text, tag)
```

Conclusion

The main goal of the master's thesis is an introduction to quantum and post-quantum cryptography, different types of both principles and analyzing each approach and their implementation of hybrid key distribution.

The theoretical part of the thesis is to introduce quantum cryptography, its quantum key distribution (QKD) and one of the most popular protocols BB84 and Coherent One-Way protocol (COW). Post-quantum cryptography introduces its different approaches, Lattice-based cryptography and problems based around this topic, multivariate, hash-based and code-based cryptography. NIST competition provided us with a big insight into possible solutions for post-quantum based cryptographical protocols. Protocols are described in detail in their relevant chapters. There are some long–standing protocols such as McEliece or new ones like NTRU or SABER. Each algorithm is based on their relevant post-quantum problems, making them good competitors to each other. The key combination is one of the most important parts of this thesis. The main goal of this thesis is a combination of two keys, quantum and post–quantum, and securely combine them into one key that way, when one of the mechanisms is weakened or broken, the other will hold the key secure to the point, when countermeasures could be done. Some of the common possible solutions are using a simple XOR function, hashing two keys together by a secure algorithm or by using a more advanced key derivation function, in this case, Extract and Expand key derivation function (HKDF) or KECCAK Message Authentication Code (KMAC) which suits this purpose ideally.

Implemented proof of concept for this problem of combining these two keys is implemented in programming language Python. Both clients were set up in internal network in Brno University of Technology (BUT). Clients were virtualized in internal laboratory. Library *PQcrypto* is used to negotiate post-quantum key while quantum key was downloaded from QKD servers using HTTPS interface. This key was negotiated through quantum channel with described properties. Key combination can be done in three different methods, those being not recommended XOR function, Hash combination with SHA–2 or SHA–3, or using advanced method for key derivation KMAC. Furthermore, a secret file is encrypted with hybrid key and sent through TCP connection to the other client that decrypted this file and saved it. Output console application is without GUI that can be ran with different starting control arguments. User manual is also attached to simple navigation and running of said demonstrator.

The main goals of this master's theses were met. Both quantum and post–quantum algorithms were analysed. Possible key combination were described and implemented in virtual environment to demonstrate creation of hybrid key.

Bibliography

[1] STEANE, Andrew. Quantum computing. Reports on Progress in Physics. 1998, 61(2), 117–173. Available from URL: <https://doi.org/10.1088/0034-4885/61/2/002>.

[2] LOTKENHAUS, N. Quantum key distribution. EQEC '05. European Quantum Electronics Conference, 2005 [online]. IEEE, 2005, 295-295 [cit. 2021-11-18]. ISBN 0-7803-8973-5. Available from: doi: 10.1109/EQEC.2005.1567461

[3] AIZAN, Nur Hanani Kamarul, Zuriati Ahmad ZUKARNAIN, Hishamuddin ZAINUDDIN, Yann THOMA and Hugo ZBINDEN. Implementation of BB84 Protocol on 802.11i. 2010 Second International Conference on Network Applications, Protocols and Services [online]. IEEE, 2010, 2010, 130-134 [cit. 2021-11-18]. ISBN 978-1-4244-8048-7. Available from: doi: 10.1109/NETAPPS.2010.31

[4] AHMED I., Khaleel. Coherent one-way protocol: Design and simulation. 2012 International Conference on Future Communication Networks [online]. 2012, 170-174 [cit. 2021-11-17]. Available from URL: <doi:10.1109/ICFCN.2012.6206863

[5] STUCKI, Damien, Sylvain FASEL, Nicolas GISIN, Yann THOMA and Hugo ZBINDEN. Coherent one-way quantum key distribution. Proc SPIE [online]. 2007 [cit. 2021-11-17]. Available from: doi:10.1117/12.722952

[6] STUCKI, Damien, Claudio BARREIRO, Sylvain FASEL, et al. High speed coherent one-way quantum key distribution prototype. Optics Express [online]. Geneva, Switzerland, 2008, (17), 9 [cit. 2021-11-17]. Available from: doi: 10.1364/OE.17.013326

[7] CHEN, Lily, Stephen JORDAN, Yi-Kai LIU, Dustin MOODY, Rene PERALTA, Ray PERLNER and Daniel SMITH-TONE. Report on Post-Quantum Cryptography [online]. 2016, 1–3 [cit. 2021-11-17]. Available from URL: <http://dx.doi.org/10.6028/NIST.IR.8105

[8] BERNSTEIN, D. a T. LANGE. Post-quantum cryptography. Nature [online]. 2017, (549), 188–194 [cit. 2021-11-17]. Available from URL: <https://doi.org/10.1038/nature23461>

[9] MICCIANCIO, Daniele and Oded REGEV. Lattice-based Cryptography [online]. Berlin, Heidelberg, 2009, 147–191 [cit. 2021-11-17]. ISBN 978-3-540-88702-7. Available from: doi:10.1007/978-3-540-88702-7_5

[10] BERNSTEIN, Daniel J. Post-Quantum Cryptography: Introduction to post-quantum cryptography [online]. Berlin, Heidelberg, Springer Berlin Heidelberg, 2009, 1–14 [cit. 2021-11-17]. ISBN 978-3-540-88702-7. Available from: doi: 10.1007/978-3-540-88702-7_1

[11] NIST Post-Quantum Cryptography Standardization NIST Information Technology Laboratory COMPUTER SECURITY RESOURCE CENTER [online], 2020 [cit. 2021-11-17]. Available from: <https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions>

[12] ALBRECHT, Martin R., Daniel J. BERNSTEIN, Tung CHOU, et al. Classic McEliece: conservative code-based cryptography [online]. 2010,6–20 [cit. 2021-11-17]. Available from URL: <https://classic.mceliece.org/nist/mceliece-20201010.pdf>

[13] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Specification document (update from August 2021) [online]. 4–15 2021-08-04 [cit. 2021-11-17] Available from URL: <https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf>

[14] CHEN, Cong, Oussama DANBA, Jefrey HOFSTEIN a Andreas HÜLSING. Algorithm Specifications And Supporting Documentation [online]. March 30, 2019 [cit. 2022-04-23]. Available from URL: <https://ntru.org/f/ntru-20190330.pdf>

[15] HOFFSTEIN, Jeffrey, Jill PIPHER a Joseph H. SILVERMAN. NTRU: A Ring-Based Public Key Cryptosystem [online]. Springer, Berlin, Heidelberg, 2006 [cit. 2022-04-23]. ISBSN 978-3-540-64657-0. Available from URL: <https://doi.org/10.1007/BFb0054868>

[16] BASSO, Andrea, Jose Maria Bermudo MERA, Jan-Pieter D'ANVERS a Angshuman KARMAKAR. SABER: Mod-LWR based KEM (Round 3 Submission) [online]. 2019 [cit. 2022-04-23]. Available from URL: <https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf>

[17] LEWIN, Michael. All About XOR. Accu [online]. 2012 [cit. 2021-11-17]. Available from URL: <https://accu.org/journals/overload/20/109/lewin_1915/

[18] AL-KUWARI, Saif; DAVENPORT, James H.; BRADFORD, Russell J. Cryptographic Hash Functions: Recent Design Trends and Security Notions. IACR

Cryptol. ePrint Arch., 2011 [cit. 2021-11-17], Available from URL: <https://eprint.iacr.org/2011/565.pdf

[19] IEEE Standard for Identity-Based Cryptographic Techniques using Pairings. IEEE Std 1363.3-2013 [online]. IEEE, 2013, 1-151 [cit. 2021-11-17]. Available from: doi: 10.1109/IEEESTD.2013.6662370

[20] BARKER, Elaine, Lily CHEN, Allen ROGINSKY, Apostol VASSILEV and Richard DAVIS. Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography [online]. 2018 [cit. 2021-11-18]. Available from: doi: 10.6028/NIST.SP.800-56Ar3

[21] KELSEY, John, Bruce SCHNEIER, Chris HALL and David WAGNER. Secure applications of low-entropy keys [online]. Berlin, Heidelberg, 1998, 121–134 [cit. 2021-11-17]. ISBN 978-3-540-69767-1. Available from: doi: 10.1007/BFb0030415

[22] Salted Password Hashing - Doing it Right. Crack Station [online]. 2021 [cit. 2021-11-17]. Available from URL: <https://crackstation.net/hashing-security.htm>

[23] KRAWCZYK, H. and P. ERONEN. HMAC-based Extract-and-Expand Key Derivation Function. Internet Engineering Task Force [online]. 2010, 1–6 [cit. 2021-11-17]. ISSN 2070-1721. Available from URL: <https://www.rfc-editor.org/rfc/rfc5869>

[24] KELSEY, John, Shu-Jen H CHANG and Ray PERLNER. SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash, 2016. NIST Special Publication [online]. 800–185 [cit. 2022-04-04]. Available from URL: <https://doi.org/10.6028/NIST.SP.800-185>

[25] pq-crypto, 2020. kpdemetriou. GitHub [online]. 8 September 2020. [cit. 2021-12-2]. Available from URL: <https://github.com/kpdemetriou/pqcrypto>

[26] NIST. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions [online]. 2015 [cit. 2021-12-05]. Available from URL: <http://dx.doi.org/10.6028/NIST.FIPS.202>

[27] NIST. Secure Hash Standard (SHS) [online]. 2015 [cit. 2021-12-05]. Available from URL:<http://dx.doi.org/10.6028/NIST.FIPS.180-4>

Symbols and abbreviations

HTTPS	Hypertext Transfer Protocol Secure
CA	Certification authority
QKD	Quantum key distribution
KEM	Key encapsulation method
COW	Coherent One-Way Protocol
RSA	Rivest–Shamir–Adleman
DH	Diffie-Hellman
SHA	Secure hash algorithm
DSA	Digital Signature Algorithm
SVP	Shortest Vector problem
CVP	Closest Vector problem
NP	Non polynomial
NIST	National Institute for Standards and Technologies
LWE	Learning-with-errors
Pk	Public key
Sk	Secret key
MD5	Message-Digest algorithm
KDF	Key derivation function
WPA	Wi-fi protected Access
HKDF	Extract-and-Expand Key Derivation Function
KMAC	KECCAK Message Authentication Cod
PRF	Pseudo-random function
IETF	Internet Engineering Task Force
HMAC	Hash-based message authentication code

API Application Programming Interface

List of appendices

A User manual

User manual for operating key combination script

This manual covers the operation of a sample script for combining quantum and post-quantum keys using selected methods. The script as well as its parts are written in Python 3.9 programming language with the help of a supporting script in bash. Once combined, the output key is used to encrypt using AES–256 symmetric cipher in GCM mode and send to the other party in the communication. The latter decrypts and saves the file. All necessary files and certificates are stored directly on the Alice and Bob clients.

A.1 Topology

The individual devices are in the internal laboratory network of the BUT in the `192.168.10.X` address range. Visualised demonstration devices Alice and Bob with Debian operating system.

Name	IP address	Login	Password
Alice	192.168.10.2	mystify	dpVUT22
Bob	192.168.10.7	mystify	dpVUT22

The quantum key establishment devices are physical devices from the Swiss manufacturer IDQ. The most essential elements for demonstration purposes are the devices shown in the table below. The devices are connected to using the SSH protocol, so certificates for each device are also required as well as keys and a certificate authority certificate.

Name	IP address	Certificate	Password to certificate
Alice (QKDA)	192.168.10.102	ENCA-cert.pem	ENCA-key.pem
Bob (QKDB)	192.168.10.107	ENCB-cert.pem	ENCB-key.pem

A.2 Dependencies

Dependencies needed to run the script:

- Python 3.9: library `Pycryptodomex` and `PqCrypto`,
- Linux package `curl` and `jq`.

A.3 Script

All the necessary files are pre-loaded for each of Alice and Bob's clients in the **/home/mystify/hybrid_key**. These files are:

- Certificates and their additional files for quantum key establishment devices, see A.1,
- Script and it parts: `client.py, master.py, slave.py, QKDscript`.

The script is run using the `client.py` file with additional parameters, all of which are mandatory:

- --initComm or --listener - defining if client will initiate communication or rather wait with open port for initiation respectively,
- --dest [ADDR] - defines destination IPv4 address (without port),
- --local [ADDR] - defines local IPv4 address of exiting interface (any client can have multiple exit interfaces, thus this functionality),
- --mPath [PATH] - defines path to file that initialising client want to send (only initialising client takes advantage of this parameter),
- --sPath [PATH] - defines save path for received file for listening client (only listening client takes advantage of this parameter),
- --comb [xor/sha2/sha3/kmac] - declares mechanism for creating hybrid key, either by XOR-based combination, Hash-based combination (SHA–2–256 or SHA–3–256) or key derivation by KMAC, respectively.

A.4 Example

Before running the script its mandatory to enter directory `\home\mystify\home\hybrid_key` by simple command `cd hybrid_key`. Afterwards its possible to start the script. Example command for Bob:
`python3 client.py -listener -dest 192.168.10.2 -local 192.168.10.7 -sPath /home/mystify/hybrid_key/secret_recovered_2 -comb xor`.
Example command for Alice:
`python3 client.py -initComm -dest 192.168.10.7 -local 192.168.10.2 -mPath /home/mystify/hybrid_key/secret -comb xor`.

Once started, a connection is established with each party. Using the PQCrypto library, a key establishment is performed on the symmetric post-quantum cipher. Alice then downloads the quantum key and an identification number, which she then sends to Bob. He downloads the same key as well. The combination of the key according to the selected method, encryption of the selected file and transmission takes place. Output for Alice is in picture A.1 and for Bob in picture A.2.

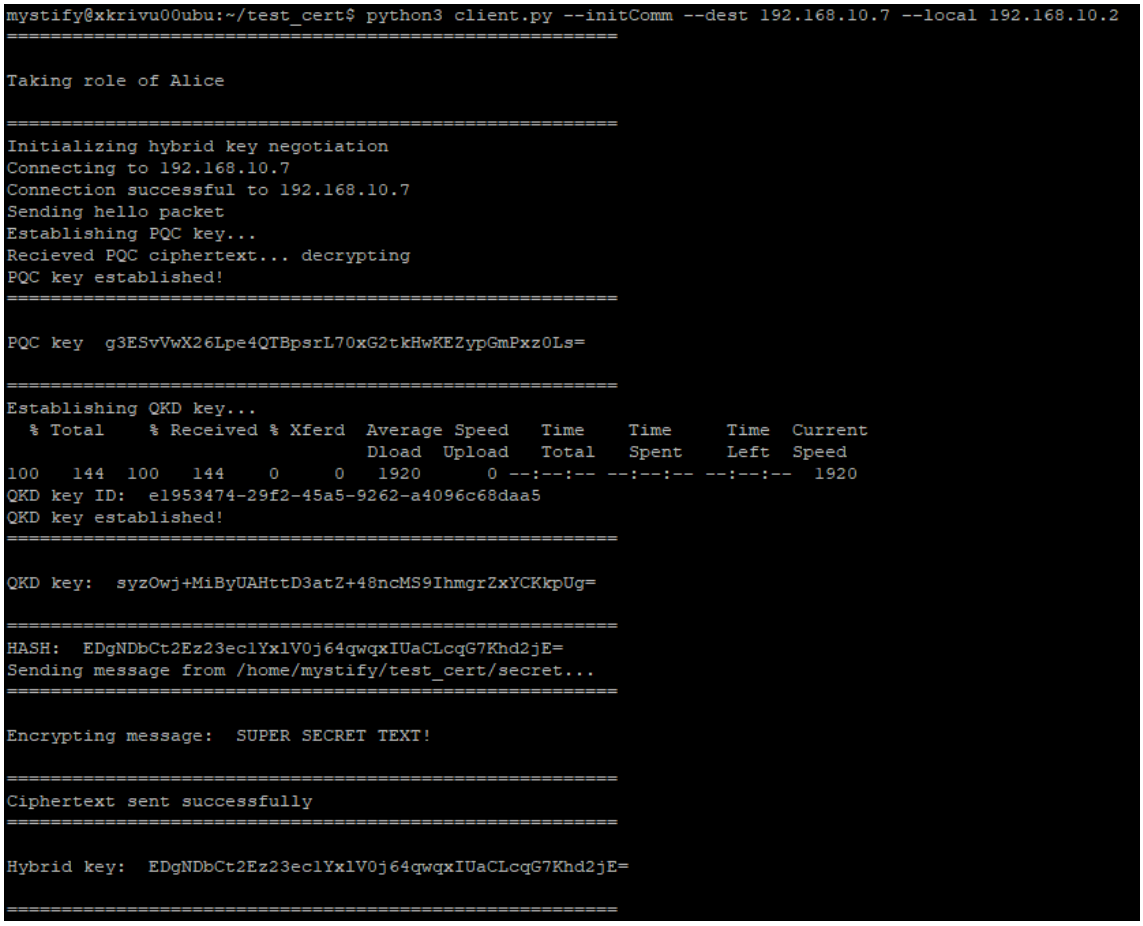

Fig. A.1: Script output on Alices side

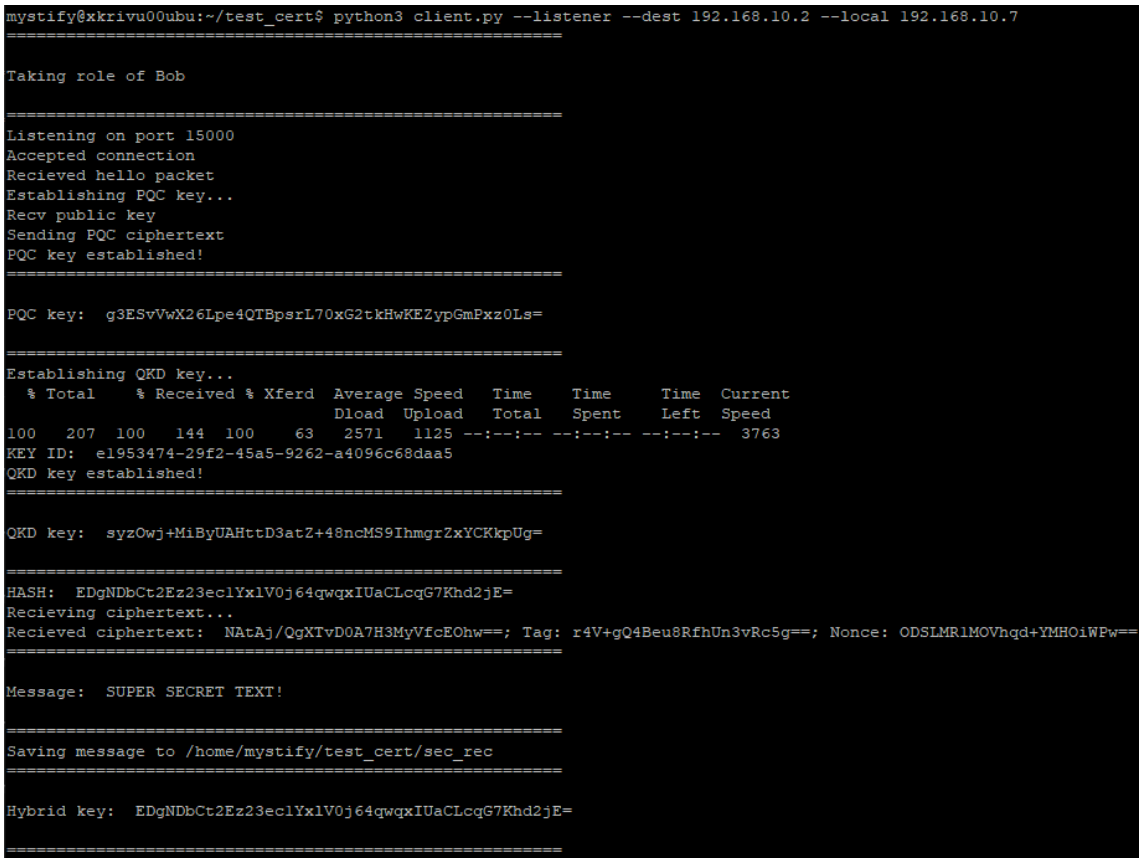

Fig. A.2: Script output on Bobs side