Post-Quantum Cryptographic Systems based on Quaternions

Cláudia da Costa Maia
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2024

Post-Quantum Cryptographic Systems based on Quaternions

Cláudia da Costa Maia

Dissertação realizada no âmbito do Mestrado de Segurança Informática

Departamento de Ciência de Computadores

2024

Orientador
Rogério Ventura Lages dos Santos Reis, Professor Auxiliar,
Faculdade de Ciências da Universidade do Porto

Coorientador
António José de Oliveira Machiavelo, Professor Auxiliar,
Faculdade de Ciências da Universidade do Porto

Acknowledgements

First and foremost, I would like to express my heartfelt gratitude to my academic advisors for allowing me to explore this fascinating topic, which is deeply intertwined with mathematics. Their unwavering support and guidance have been invaluable throughout this journey.

I am deeply grateful to my parents and my sister for their continuous support and encouragement. Special thanks also go to my boyfriend and my friends for their relentless motivation and strength, especially during the more challenging times. Their encouragement has been crucial in helping me persevere, even while working and facing difficult periods.

Thank you all who have been part of this journey.

Abstract

The development of quantum technology poses significant risks to classical security mechanisms. With these more powerful computers, it becomes possible to drastically reduce the execution times of certain computational problems, rendering them trivial to solve. Given that many current cryptographic methods are vulnerable to quantum attacks, it is crucial to explore alternative methods to safeguard information as quantum computers evolve. The widespread use of cryptographic techniques that may soon be obsolete needs proactive research into more secure solutions.

In the context of quantum computing, cryptographic methods that rely on integer factorization or discrete logarithms, such as RSA, which is widely used, become vulnerable to much faster attacks. This significantly diminishes their security level. Quantum algorithms, like Shor's algorithm, can solve these problems in polynomial time, which classical computers cannot achieve. As a result, the robustness of these widely trusted cryptographic systems is compromised, highlighting the urgent need for new cryptographic strategies that can withstand quantum attacks.

To contribute to the development of more secure methods against quantum computers, we reviewed several research papers on quaternion-based cryptography, which are resistant to quantum attacks.

Keywords: Quaternions; Modular arithmetic; Lattices; Probabilities; Post-Quantum Cryptography; Diffie-Hellman; NTRU; QTRU.

Resumo

O aparecimento da tecnologia quântica coloca riscos significativos aos mecanismos de segurança clássicos. Com estes computadores mais poderosos, torna-se possível reduzir drasticamente os tempos de execução de certos problemas computacionais, tornando-os triviais de resolver. Dado que muitos métodos criptográficos atuais são vulneráveis a ataques quânticos, é crucial explorar métodos alternativos para salvaguardar a informação à medida que a tecnologia quântica evolui. O uso generalizado de técnicas criptográficas que em breve poderão estar obsoletas necessita de investigação proativa para soluções mais seguras.

No contexto da computação quântica, métodos criptográficos que se baseiam na fatorização de inteiros ou logaritmos discretos, como o RSA, que é amplamente utilizado, tornam-se vulneráveis a ataques muito mais rápidos. Isto diminui significativamente o seu nível de segurança. Algoritmos quânticos, como o algoritmo de Shor, podem resolver estes problemas em tempo polinomial, algo que os computadores clássicos não conseguem alcançar. Como resultado, a robustez destes sistemas criptográficos amplamente confiáveis é comprometida, destacando a necessidade urgente de novas estratégias criptográficas que possam resistir a ataques quânticos.

Para contribuir para o desenvolvimento de métodos mais seguros contra computadores quânticos, estudamos vários artigos de investigação sobre criptografia baseada em quaterniões, que são resistentes a ataques quânticos.

Palavras-chave: Quaterniões; Aritmética modular; Reticulados; Probabilidades; Criptografia Pós-Quântica; Diffie-Hellman; NTRU; QTRU.

Contents

Listings

Acronyms

DCC Departamento de Ciência de Computadores

FCUP Faculdade de Ciências da Universidade do Porto

RSA Rivest-Shamir-Adleman

DSA Digital Signature Algorithm

ECDSA Elliptic Curve Digital Signature Algorithm

DES Data Encryption Standard

3DES Triple Data Encryption Standard

AES Advanced Encryption Standard

AES-128/256 Advanced Encryption Standard (128-bit/256-bit)

RC4 Rivest Cipher 4

ECC Elliptic-Curve Cryptography

DH Diffie-Hellman

ECDH Elliptic-Curve Diffie-Hellman

PMI Post-Quantum Multivariate Interpolation

ipHFE internal perturbation Hidden Field Equations

UOV Unbalanced Oil and Vinegar

XMSS eXtended Merkle Signature Scheme

TLS Transport Layer Security

IPSEC Internet Protocol Security

SSH Secure Shell

SVP Shortest Vector Problem

CVP Closest Vector Problem

SIVP Shortest Independent Vectors Problem

CML Convolution Modular Lattices

LLL Lenstra-Lenstra-Lovász lattice basis reduction algorithm

MPKCs Multivariate Public Key Cryptosystems

MQ Multivariate Quadratic problem

NP-hard Non-deterministic Polynomial-time hard

Chapter 1

Introduction

The advancement of quantum technologies presents notable threats to traditional security protocols. Some computational problems can have their execution durations significantly shortened by using these more powerful computers, making them easy to solve. It is critical to investigate alternate techniques to protect information as quantum technology develops because many of the cryptographic techniques used now are susceptible to quantum attacks. Proactive research into more secure methods is necessary due to the extensive use of cryptographic techniques that may become obsolete shortly.

Cryptographic techniques like the widely used RSA and ECC, that depend on discrete logarithms or integer factorization, are susceptible to substantially faster attacks in the quantum computing age. Their level of security is thereby greatly reduced.

These issues can be solved in polynomial time using quantum algorithms such as Shor's algorithm, which is not possible with conventional computers. This compromises the strength of existing widely used cryptographic systems, underscoring the pressing need for new cryptographic techniques resistant to quantum attacks.

To contribute to the development of more secure defences against quantum computers, we reviewed several studies on quaternion-based cryptography, which is resilient to quantum attacks (3). To ensure the reader understands each term, we began our work by defining the relevant concepts (2). Then examined how traditional cryptographic techniques could be modified to utilize quaternions. Then we implemented the schemes to help understand them better and experiment with them (A).

This thesis is composed of an introduction, two main chapters, and a conclusion. The main chapters are structured as follows:

- **Basic Concepts**: This chapter covers all the fundamental concepts that the reader should be minimally familiar with, including definitions in algebra and probability. Then it is explained what quaternions are, their properties, and what quaternion algebras are. Additionally, it covers various concepts of classical and post-quantum cryptography.
- **Development**: Here is compiled several cryptographic schemes using quaternions that are resistant to quantum attacks. Each scheme is explained first without using quaternions and then the adaptation with quaternions. The analyzed schemes include Diffie-Hellman, its adaptation, NTRU, and its adaptation QTRU.

Chapter 2

Basic Concepts

2.1 Algebra

According to [12], **Algebra** is a fundamental branch of mathematics focused on studying **algebraic structures** and their interrelations. It entails the manipulation and investigation of symbols and mathematical operations to solve equations and characterize the properties of mathematical entities. Algebra covers a broad spectrum of subjects, including elementary operations (addition, subtraction, multiplication, division), linear and quadratic equations, systems of equations, matrices, and determinants, as well as advanced topics such as group theory, ring theory, and field theory. Subsequently, we will recall some basic algebraic structures, necessary for comprehending the topics addressed in this thesis.

2.1.1 Group

A **Group** is a nonempty set G equipped with a binary operation denoted by $(\cdot)$, which is a function $G \times G \to G$. It satisfies the following properties:

1. **Associativity:** For all $a, b, c \in G$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
2. **(Two-sided) Identity Element:** There exists an element e in G such that for all $a \in G$, $a \cdot e = e \cdot a = a$.
3. **(Two-sided) Inverse Element:** For each a in G, there exists an element a^{-1} in G such that $a \cdot a^{-1} = a^{-1} \cdot a = e$, where e is the identity element.

An **Abelian Group** is a Group G with the property of **Commutativity**: For all $a, b \in G$, $a \cdot b = b \cdot a$.

2.1.2 Ring

A **Ring** is a nonempty set R together with two binary operations (usually called as addition and multiplication, usually denoted by $(+)$ and $(\cdot)$ respectively) satisfying the following properties:

1. **Additive Group:** $(R, +)$ forms an Abelian Group. So:
   (a) $\forall a, b, c \in R$, $(a + b) + c = a + (b + c)$
   (b) $\exists 0_R \in R, \forall a \in R : a + 0_R = 0_R + a = a$
   (c) $\forall a \in R, \exists -a \in R : a + (-a) = (-a) + a = 0_R$
   (d) $\forall a, b \in R : a + b = b + a$
2. **Associative Multiplication:** $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c \in R$.
3. **(Two-sided) Distributive Laws:** $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(a + b) \cdot c = a \cdot c + b \cdot c$.

Additionally:

1. R Is said to be a **Commutative Ring** if: $\forall a, b \in R : a \cdot b = b \cdot a$.
2. R Is said to be a **Ring with Identity** if: $\exists 1_R \in R : 1_R \cdot a = a \cdot 1_R = a, \forall a \in R$.

2.1.2.1 Non-Commutative Ring

A **Non-Commutative Ring** is a Ring in which there exist elements a, b in R such that $a \cdot b \neq b \cdot a$.

2.1.2.2 Division Ring

A nonzero element a in a ring R is said to be a **left (resp. right) zero divisor** if there exists a nonzero $b \in R$ such that $a \cdot b = 0_R$ (resp. $b \cdot a = 0_R$). A **zero divisor** is an element of R which is both a left and a right zero divisor.

An element a in a ring R with identity is said to be **left (resp. right) invertible** if there exists $c \in R$ (resp. $b \in R$) such that $c \cdot a = 1_R$ (resp. $a \cdot b = 1_R$). The element c (resp. b) is called a **left (resp. right) inverse** of a. An element $a \in R$ that is both left and right invertible is said to be **invertible** or to be a **unit**.

A commutative ring R with identity $1_R \neq 0_R$ and no zero divisors is called an **integral domain**.

A ring D with identity $1_D \neq 0_D$ in which every nonzero element is a unit is called a **division ring**.

An **Euclidean division ring** is a ring R equipped with a Euclidean function $\phi : R \setminus \{0_R\} \to \mathbb{N}$ such that for any elements a and b in R with $b \neq 0_R$, there exist elements q (quotient) and r (remainder) in R such that $a = bq + r$, and either $r = 0_R$ or $\phi(r) < \phi(b)$.

2.1.3 Field

A **Field** F is a **commutative division ring**. That means $\forall a, b \in F, a \cdot b = b \cdot a$.

The **characteristic of a field** F, denoted as char(F), is the smallest positive integer p such that the sum of p copies of the field's multiplicative identity is equal to 0_F. In other words, char(F) is the number of times we need to add the field's identity element to itself to get zero. If p does not exist, the ring is said to have characteristic zero. It is not hard to show that p must be a prime number.

A **skew field** is a division ring, where multiplication is not necessarily commutative.

2.1.4 Ideal

Let R be a ring and S a nonempty subset of R, that is closed under the operations of addition and multiplication in R. If S forms a ring under these operations, then S is a **subring** of R. When R is a ring with identity, one also requires S to have the same identity in order for it to be a subring of R.

A subring I of a ring R is a **left (resp. right) ideal** if:

$$\forall r \in R, \forall x \in I \Rightarrow r \cdot x \in I (resp.\ x \cdot r \in I).$$

It can be shown that, a nonempty subset I of a ring R is a **left (resp. right) ideal** if and only if $\forall a, b \in I, \forall r \in R$:

1. $a, b \in I \Rightarrow a - b \in I$;
2. $a \in I, r \in R \Rightarrow r \cdot a \in I (resp.\ a \cdot r \in I)$.

I is an **Ideal** if it satisfies both left and right ideal properties.

A group is **generated** by a set of elements if all the elements of the group can be constructed from combinations of the elements of this set, using the group operation. We say that a number $a \in R$ **generates** R if the set $\{a^n \mid n \in \mathbb{Z}\}$ is equal to R.

For a ring R and an element $a \in R$:

- The **principal left ideal** generated by a is the set: $\langle a \rangle = \{r \cdot a \mid r \in R\}$.
- The **principal right ideal** generated by a is the set: $\langle a \rangle = \{a \cdot r \mid r \in R\}$.
- The **principal ideal** usually refers to: $\langle a \rangle = \{r \cdot a \cdot s \mid r, s \in R\}$.
  In a commutative ring, this simplifies to: $\langle a \rangle = \{r \cdot a \mid r \in R\}$, as multiplication is commutative.

A **principal ideal** is an ideal that is generated by a single element, and is the smallest ideal in R that contains that element.

2.1.5 Modular Arithmetic

Modular Arithmetic is a branch of number theory that deals with operations performed on remainders. If $a \equiv b \pmod{m}$, it means that a is congruent to b modulo m, indicating that a and b have the same remainder when divided by m ($\Leftrightarrow m|a-b$).

2.1.6 Lattices

Suppose $v_1, \ldots, v_n \in \mathbb{R}^m$ form a set of linearly independent vectors. The **lattice** L generated by $v_1, \ldots, v_n$ is defined as the set of all linear combinations of $v_1, \ldots, v_n$ with integer coefficients:

$$L = \{a_1 v_1 + a_2 v_2 + \ldots + a_n v_n : a_1, a_2, \ldots, a_n \in \mathbb{Z}\}.$$

A **basis** for L is any set of independent vectors that generate L. Any two such sets have the same number of elements. The dimension of L is defined as the number of vectors in a basis for L.

An **integral (or integer) lattice** is defined as a lattice in which all vectors have integer coordinates. Equivalently, an integral lattice can be described as an additive subgroup of $\mathbb{R}^m$ for some $m \geq 1$ which is isomorphic to $\mathbb{Z}^m$.

2.2 Probabilities

In this section, we will recall some key concepts of probability. The notion of probability is an ancient concept that has been interpreted in various ways over time. However, the classical definition, by Laplace in 1812, says that in a random experiment with m possible outcomes, all equally likely, if k of these outcomes lead to the occurrence of a specific event, the probability of an event to be A is defined classically as:

$$P(X = A) = \frac{k}{m}$$

where, P is the probability function, $X = A$ indicates the event X being A, k is the number of favourable outcomes, and m is the total number of possible outcomes.

A **random variable** is a measurable function $X : \Omega \to E$ that maps each outcome $\omega \in \Omega$ to a number $X(\omega) \in E$. Where Ω is the sample space, which represents the set of all possible

outcomes of a random experiment, and E is the state space or image space, which represents the set of all possible values that the random variable X can take based on the outcomes in Ω.

2.2.1 Expected Value

If a discrete random variable X has $x_1, x_2, \ldots$ as possible values, and its probability function is P, the **expected value** (or **mathematical expectation** or **mean**) of X is given by:

$$E(X) = \sum_i x_i \, P(X = x_i)$$

provided that $\sum_i |x_i| \, P(X = x_i)$ is finite. This represents the **weighted average** of all possible outcomes, where the weights are given by the probabilities of those outcomes occurring.

Some properties seen in [9] are:

- $E(c) = c$ for a constant c.
- $E(cX + b) = c\,E(X) + b$ for constants c and b.

2.2.2 Standard Deviation and Variance

The **variance** of a random variable X, if it exists, is given by:

$$\sigma^2 = \mathrm{Var}(X) = E((X - E(X))^2) = E(X^2) - E^2(X).$$

It measures how far the values of the random variable are spread out from their expected value.

The **standard deviation** of a random variable X is the positive square root of the variance:

$$\sigma = \sqrt{\mathrm{Var}(X)}.$$

Some properties in [9]:

- $\mathrm{Var}(c) = 0$ for a constant c.
- $\mathrm{Var}(cX + b) = c^2\mathrm{Var}(X)$ for constants c and b.

2.2.3 Covariance

Two events A and B are said to be **independent** if the knowledge of one does not influence the probability of the other occurring. That is, the probability of A knowing B is equal to the probability of A ($P(A \mid B) = P(A)$).

Covariance measures the degree to which two random variables change together. For two random variables X and Y, the covariance $\text{Cov}(X, Y)$ is defined as:

$$\text{Cov}(X, Y) = E[(X - E(X)) \cdot (Y - E(Y))]$$

where $E(X)$ and $E(Y)$ are the expected values (means) of X and Y, respectively.

The sign of the covariance indicates the tendency of X and Y to vary together:

- $\text{Cov}(X, Y) > 0$: X and Y tend to increase or decrease together.
- $\text{Cov}(X, Y) < 0$: X tends to increase when Y decreases, and vice versa.
- $\text{Cov}(X, Y) = 0$: X and Y are uncorrelated and independent, meaning their variations are independent of each other.

For two independent random variables X and Y, the following properties hold, as referred in [9]:

- $\text{Cov}(X, Y) = 0$;
- $E(XY) = E(X) \cdot E(Y)$;
- $E((XY)^2) = E(X^2) \cdot E(Y^2)$;
- $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$;
- $\text{Var}(XY) = \text{Var}(X) \cdot \text{Var}(Y) + E^2(Y) \cdot \text{Var}(X) + E^2(X) \cdot \text{Var}(Y)$.

The last two properties can be obtained as follows:

$$\begin{aligned}
\text{Var}(X + Y) &= E((X + Y)^2) - E^2(X + Y) \\
&= E(X^2 + 2 \cdot X \cdot Y + Y^2) - (E(X) + E(Y))^2 \\
&= E(X^2) + 2 \cdot E(X) \cdot E(Y) + E(Y^2) - E(X)^2 - 2 \cdot E(X) \cdot E(Y) - E(Y)^2 \\
&= E(X^2) - E(X)^2 + E(Y^2) - E(Y)^2 \\
&= \text{Var}(X) + \text{Var}(Y).
\end{aligned}$$

We have:

$$\text{Var}(XY) = E((XY)^2) - E^2(XY) = E(X^2) \cdot E(Y^2) - E^2(X) \cdot E^2(Y)$$

And,

$$\begin{aligned}\mathsf{Var}(X)\mathsf{Var}(Y) &= (E(X^2) - E^2(X)) \cdot (E(Y^2) - E^2(Y)) \\ &= E(X^2) \cdot E(Y^2) - E^2(X) \cdot E(Y^2) - E(X^2) \cdot E^2(Y) + E^2(X) \cdot E^2(Y) \\ &= \mathsf{Var}(XY) + E^2(X) \cdot E^2(Y) - E^2(X) \cdot E(Y^2) - E(X^2) \cdot E^2(Y) + E^2(X) \cdot E^2(Y) \\ &= \mathsf{Var}(XY) + E^2(Y) \cdot (E^2(X) - E(X^2)) + E^2(X) \cdot (E^2(Y) - E(Y^2)) \\ &= \mathsf{Var}(XY) - E^2(Y) \cdot \mathsf{Var}(X) - E^2(X) \cdot \mathsf{Var}(Y)\end{aligned}$$

So, we get:

$$\mathsf{Var}(XY) = \mathsf{Var}(X) \cdot \mathsf{Var}(Y) + E^2(Y) \cdot \mathsf{Var}(X) + E^2(X) \cdot \mathsf{Var}(Y).$$

2.3 Quaternion Algebras

According to the book "Quaternion Algebras" [23] by John Voight, an **algebra** over a field $\mathbb{F}$ is a ring $\mathbb{A}$ equipped with a homomorphism $\mathbb{F} \to \mathbb{A}$ such that the image of $\mathbb{F}$ lies in the centre $Z(\mathbb{A})$ of $\mathbb{A}$, defined by:

$$Z(\mathbb{A}) = \{\alpha \in \mathbb{A} : \alpha \cdot \beta = \beta \cdot \alpha \text{ for all } \beta \in \mathbb{A}\};$$

if $Z(\mathbb{A}) = \mathbb{F}$, then we say $\mathbb{A}$ is central (as an $\mathbb{F}$-algebra). Another way to define a $\mathbb{F}$-algebra is to let $\mathbb{F}$ be a field and $\mathbb{A}$ a vector space over $\mathbb{F}$ equipped with an additional binary operation $\cdot$ from $\mathbb{A} \times \mathbb{A}$ to $\mathbb{A}$. Then $\mathbb{A}$ is an algebra over $\mathbb{F}$ if the following identities hold for all elements $x, y, z \in \mathbb{A}$, and all elements $a, b \in \mathbb{F}$:

- $(x + y) \cdot z = x \cdot z + y \cdot z$
- $z \cdot (x + y) = z \cdot x + z \cdot y$
- $(ax) \cdot (by) = (ab)(x \cdot y)$.

A quaternion algebra over a field $\mathbb{F}$ is a central algebra that is a 4-dimensional vector space over $\mathbb{F}$, with basis $\{1, i, j, k = ij\}$. This was firstly introduced by Sir William Rowan Hamilton. For years, Hamilton sought to generalize complex numbers to three dimensions. His breakthrough came when he realized that a fourth dimension was needed, leading to the formulation of quaternions. On October 16, 1843, Sir William Rowan Hamilton carved the equations

$$i^2 = j^2 = k^2 = ijk = -1$$

into Brougham Bridge in Dublin, marking the discovery of quaternions.

A **Hamiltonian quaternion** is typically written as

$$q = q_0 + q_1 i + q_2 j + q_3 k, \tag{2.1}$$

where q_0, q_1, q_2, and q_3 are real numbers, and i, j, and k are imaginary units satisfying the following relations:

$$i^2 = j^2 = k^2 = ijk = -1 \text{ and } ij = -ji = k, \tag{2.2}$$

which completely determine the **multiplication** of quaternions:

$$\begin{aligned}(q_0 + q_1 i + q_2 j + q_3 k)(v_0 + v_1 i + v_2 j + v_3 k) = \\ (q_0 v_0 - q_1 v_1 - q_2 v_2 - q_3 v_3) + (q_0 v_1 + q_1 v_0 + q_2 v_3 - q_3 v_2) i + \\ (q_0 v_2 - q_1 v_3 + q_2 v_0 + q_3 v_1) j + (q_0 v_3 + q_1 v_2 - q_2 v_1 + q_3 v_0) k.\end{aligned} \tag{2.3}$$

The set of Hamiltonian quaternions over $\mathbb{R}$, is denoted by $\mathbb{H}$, and $\mathbb{H}(\mathbb{Q})$ denotes the Hamiltonian quaternion division algebra over $\mathbb{Q}$.

The symbol $\mathcal{L}$ denotes the collection of integral Hamiltonian quaternions, also known as **Lipschitz integers**. It constitutes the set:

$$\{a + bi + cj + dk \in \mathbb{H} : a, b, c, d \in \mathbb{Z}\}. \tag{2.4}$$

$\mathcal{H}$ denotes the ring of **Hurwitz integers**, defined as the union of $\mathcal{L}$ and $\omega + \mathcal{L}$, where $\omega = \frac{1}{2}(1 + i + j + k)$. This set is given by:

$$\left\{\frac{a}{2} + \frac{b}{2} i + \frac{c}{2} j + \frac{d}{2} k \in \mathbb{H} : a, b, c, d \in \mathbb{Z} \text{ and } a \equiv b \equiv c \equiv d \pmod 2\right\} \tag{2.5}$$

With the ring $\mathcal{H}$ of Hurwitz integers, we gain the ability to apply the Euclidean algorithm for quaternions. This algorithm allows us to compute the greatest common divisors and perform division with remainders, similar to integer arithmetic.

The **conjugate** of a quaternion q is defined as:

$$\bar{q} = q_0 - q_1 i - q_2 j - q_3 k.$$

The function $N : \mathbb{H} \to \mathbb{R}$ is called **norm**, defined as $N(x) = x\bar{x}$. Additionally, $Tr : \mathbb{H} \to \mathbb{R}$ represents the **trace** map given by $Tr(x) = x + \bar{x}$.

From the definition of quaternions, they are **noncommutative**, as for example $ij \neq ji$.

For all $q \in \mathbb{H}$, if $q \neq 0$, then $N(q) \neq 0$ and so, $q^{-1} = \frac{\bar{q}}{N(q)}$ is its inverse. Thus, as every nonzero

element in $\mathbb{H}$ has an inverse, $\mathbb{H}$ is a division ring. The Hamiltonian quaternions are, indeed, a **noncommutative division ring**, the first of its sort discovered.

Given $u = u_0 + u_1 i + u_2 j + u_3 k \in \mathbb{H}$, one defines:

$$\begin{aligned} \mathcal{R}(u) &:= \frac{1}{2}(u + \bar{u}) = u_0, \text{ the real part of } u \\ \mathcal{V}(u) &:= \frac{1}{2}(u - \bar{u}) = u_1 i + u_2 j + u_3 k, \text{ the vector part of } u. \end{aligned}$$

A quaternion u is said to be **pure** when $\mathcal{R}(u) = 0$. The set of pure quaternions is a vector space of dimension 3. And from (2.3), if $v = v_0 + v_1 i + v_2 j + v_3 k \in \mathbb{H}$, we get:

$$\mathcal{R}(uv) = u \bullet \bar{v}, \tag{2.6}$$

$$\mathcal{V}(u)\mathcal{V}(v) = -\mathcal{V}(u) \bullet \mathcal{V}(v) + \mathcal{V}(u) \times \mathcal{V}(v), \tag{2.7}$$

where $\bullet$ is the usual scalar product and $\times$ is the usual vector product in $\mathbb{R}^3$:

$$u \bullet v = u_0 v_0 + u_1 v_1 + u_2 v_2 + u_3 v_3$$

$$\mathcal{V}(u) \times \mathcal{V}(v) = \begin{vmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ \mathbf{i} & \mathbf{j} & \mathbf{k} \end{vmatrix} = (u_2 v_3 - u_3 v_2)\mathbf{i} - (u_1 v_3 - u_3 v_1)\mathbf{j} + (u_1 v_2 - u_2 v_1)\mathbf{k},$$

i.e., the determinant is computed formally using Laplace expansion along the third row.

From this it follows that $\mathbb{H} \simeq \mathbb{R} \oplus \mathbb{R}^3$, with $u \mapsto (\mathcal{R}(u), \mathcal{V}(u))$ and $(\lambda, y)(\mu, w) = (\lambda\mu - y \bullet w, \mu y + \lambda w + y \times w)$, since, for $u, v \in \mathbb{H}$:

$$\begin{aligned} uv &= \mathcal{R}(u)v + \mathcal{R}(v)u - u \bullet v + \mathcal{V}(u) \times \mathcal{V}(v) \\ &= u_0(v_0 + v_1 i + v_2 j + v_3 k) + v_0(u_0 + u_1 i + u_2 j + u_3 k) \\ &- u_0 v_0 - u_1 v_1 - u_2 v_2 - u_3 v_3 + (u_2 v_3 - u_3 v_2)i + (u_3 v_1 - u_1 v_3)j + (u_1 v_2 - u_2 v_1)k. \end{aligned} \tag{2.8}$$

One has the following relations:

$$\begin{aligned} \overline{uv} &= \mathcal{R}(v)\bar{u} + \mathcal{R}(u)\bar{v} - u \bullet v - \mathcal{V}(u) \times \mathcal{V}(v) \\ &= \mathcal{R}(\bar{u})\bar{v} + \mathcal{R}(\bar{v})\bar{u} - \bar{v} \bullet \bar{u} + \mathcal{V}(\bar{v}) \times \mathcal{V}(\bar{u}) = \bar{v}\bar{u}; \end{aligned}$$

$$u \bullet v = \Re(u\bar{v}) = \frac{1}{2}(u\bar{v} + v\bar{u});$$

$$u \bullet u = \frac{1}{2}(u\bar{u} + u\bar{u}) = u\bar{u} = N(u);$$

$$N(u + v) = (u + v)(\bar{u} + \bar{v}) = u\bar{u} + u\bar{v} + v\bar{u} + v\bar{v} = N(u) + 2\, u \bullet v + N(v).$$

Following these and (2.3, 2.8), now we have that:

$$\begin{aligned} uv &= \frac{1}{2}v(u+\bar{u}) + \frac{1}{2}u(v+\bar{v}) - \frac{1}{2}(v\bar{u}+u\bar{v}) + \mathcal{V}(u)\times\mathcal{V}(v) \\ &= \frac{1}{2}uv + \frac{1}{2}vu + \mathcal{V}(u)\times\mathcal{V}(v), \end{aligned}$$

and so,

$$uv - vu = 2\mathcal{V}(u)\times\mathcal{V}(v),$$

which shows that **two quaternions commute if and only if their vector parts are collinear.**

From (2.3) we can see that the action on $\mathbb{H}$ given by multiplication on the left by $q = q_0 + q_1 i + q_2 j + q_3 k$ corresponds to the action on $\mathbb{R}^4$ given by the matrix:

$$\begin{pmatrix} q_0 & -q_1 & -q_2 & -q_3 \\ q_1 & q_0 & -q_3 & q_2 \\ q_2 & q_3 & q_0 & -q_1 \\ q_3 & -q_2 & q_1 & q_0 \end{pmatrix}$$

because this matrix acts on the vector $v = \begin{pmatrix} v_0 & v_1 & v_2 & v_3 \end{pmatrix}$ to produce the coefficients of the product $q \cdot v$ as:

$$\begin{pmatrix} q_0 & -q_1 & -q_2 & -q_3 \\ q_1 & q_0 & -q_3 & q_2 \\ q_2 & q_3 & q_0 & -q_1 \\ q_3 & -q_2 & q_1 & q_0 \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} q_0v_0 - q_1v_1 - q_2v_2 - q_3v_3 \\ q_0v_1 + q_1v_0 + q_3v_3 - q_2v_2 \\ q_0v_2 - q_1v_1 + q_2v_0 - q_3v_3 \\ q_0v_3 + q_1v_1 - q_3v_2 + q_2v_0 \end{pmatrix}.$$

From this, one obtains an isomorphism between $\mathbb{H}$ and a subring of $M_{4\times4}(\mathbb{R})$.

On the other hand, we can write $a+bi+cj+dk$ in the form $\alpha+\beta j$ with $\alpha = a+bi, \beta = c+di \in \mathbb{C}$. And $j\alpha = aj + bji = aj - bij = \bar{\alpha}j$, so $(x+yj)(\alpha+\beta j) = (x\alpha - y\bar{\beta}) + (x\beta + y\bar{\alpha})j$. This shows that the action on $\mathbb{C}^2$ given by multiplication on the right by $\alpha+\beta j$ has matrix:

$$\begin{pmatrix} \alpha & \beta \\ -\bar{\beta} & \bar{\alpha} \end{pmatrix}$$

In this way, we get another representation of $\mathbb{H}$, as a subring of $M_{2\times2}(\mathbb{C})$.

The Hamiltonian quaternions can be **generalized** by replacing the field of real numbers $\mathbb{R}$ by any arbitrary field $\mathbb{F}$ (or ring $\mathcal{R}$). Additionally, instead of defining $i^2 = j^2 = k^2 = -1$ and $ij = -ji = k$, we define $i^2 = a, j^2 = b, k^2 = -ab$, and $ij = -ji = k$, with $a, b \in \mathbb{F}$ (or $\mathcal{R}$). These relations yields different algebras by choosing different values of a and b. It can be shown in [23], that this is

indeed a quaternion algebra, which is denoted by:

$$\mathbb{A} = \left(\frac{a,b}{\mathbb{F}}\right) = \{\alpha + \beta i + \gamma j + \delta k \in \mathbb{A} \mid \alpha, \beta, \gamma, \delta \in \mathbb{F}, i^2 = a, j^2 = b, ij = -ji = k\},$$

in fact any quaternion algebra is of this form.

In this case, we define $N(\alpha + \beta i + \gamma j + \delta k) = \alpha^2 - a\beta^2 - b\gamma^2 + ab\delta^2$. If we choose a and b to be -1 and $\mathbb{F}$ the field of real numbers $\mathbb{R}$, we get the Hamiltonian quaternion algebra, i.e., $\mathbb{H} = \left(\frac{-1,-1}{\mathbb{R}}\right)$.

From [23, 5.4.4, page 72] if we choose a, b and $\mathbb{F}$ such that for any $q \in \left(\frac{a,b}{\mathbb{F}}\right)$, $N(q) = 0$ implies $q = 0$, then $A = \left(\frac{a,b}{\mathbb{F}}\right)$ is a division ring. This property ensures that every $q \in \mathbb{A}$ is invertible, since $q^{-1} = \frac{\bar{q}}{N(q)}$, except for $N(q) = 0$ which occurs if and only if $q = 0$.

When $\mathbb{A}$ is not a division ring, it can be shown ([23]) that $\mathbb{A}$ is isomorphic to $M_2(\mathbb{F})$, the ring of all 2×2 matrices with entries from $\mathbb{F}$. This algebra is called a *split* algebra, and in this case there are some nonzero elements $q \in A$ which have norm zero, and so no multiplicative inverses.

2.4 Cryptography

Cryptography is the science and practice of secure communication in the presence of third parties (adversaries). It encompasses techniques for encrypting and decrypting information, ensuring confidentiality, integrity, and authentication of data. Cryptographic methods involve algorithms, keys, protocols, and mathematical principles to protect sensitive information.

A **cipher** is a method or algorithm used for encrypting or decrypting data to secure its confidentiality. It transforms plaintext (readable data) into ciphertext (encrypted data), which can only be decrypted back into plaintext using a specific key or algorithm.

2.4.1 Symmetric Cryptography

According to [14], a single shared secret is used in **symmetric cryptography**, also referred to as **secret key cryptography**, to enable the exchange of encrypted data between multiple parties. This type of cipher is referred to as "symmetric" since the **same key is used for both encryption and decryption**. To simplify, the sender encrypts data with a specific password, and the recipient requires the same password to retrieve the data.

Symmetric encryption is a **bidirectional** process. When provided with a block of plaintext and a designated key, symmetric ciphers consistently generate identical ciphertexts. Similarly, using the same key as input to the algorithm for decryption on a given block of ciphertext consistently yields the original plaintext. This technique is widely used for private data storage and works

well for data protection between parties that have a **pre-established shared key**. Some of the most widely used symmetric cryptography systems include **DES, 3DES, AES, and RC4**.

2.4.2 Asymmetric Cryptography

As seen in [20], **asymmetric cryptography**, also referred to as **public-key cryptography**, makes use of two keys: a **private key** that is kept secret and a **public key** that is released publicly. This method enables secure communication **without** requiring the **secret key to be shared** by both parties. With asymmetric encryption, senders uses the recipient's public key for data encryption, while the recipient uses his/her private key for decryption. This approach eliminates the requirement for secret key exchanges, and it allows the creation of digital signatures, to verify data authenticity.

Each user in asymmetric cryptography possesses a **key pair**: a public key shared openly and a private key kept secret. Both keys are **mathematically related** and are essential for cryptographic operations. The private key can be used to generate **digital signatures**, allowing the recipient to confirm the **sender's legitimacy**.

Asymmetric cryptography is frequently employed for exchanging secret keys in preparation **for symmetric cryptography** use, where one party encrypts the secret key with the recipient's public key for subsequent secure communication. Asymmetric encryption algorithms include **Elliptic Curve Cryptography (ECC), Diffie-Hellman, and RSA**.

2.4.3 Post-Quantum Cryptography

Quantum computation presents a challenge by potentially rendering many encryption methods, currently considered secure for classical computers, but are **vulnerable to efficient quantum algorithms**. In this way, post-quantum cryptography operates under the assumption that attackers possess powerful quantum computers and it aims to design cryptosystems that remain secure when faced with this scenario. Classical complexity theory doesn't provide definitive proofs of whether efficient algorithms exist for specific problems, instead, assessments of a problem's difficulty rely on empirical evidence gathered over time. Determining which problems are hard for quantum computers requires extensive, long-term research efforts by multiple researchers.

In classical computation, information is stored in bits, holding binary digits with values of 0 or 1. However, in quantum computing, the fundamental unit, called **qubits**, can hold both 0 and 1 values - known as a **superposition of two states**. The act of measuring a qubit's state causes it to "collapse into" either 0 or 1. Curiously, preparing identical strings of qubits does not always yield the same resulting bit strings, enabling quantum computers to perform rapid **parallel computations**, giving them an advantage over classical computers.

Arising from parallel computation, Shor's discovery of quantum algorithms for integer factoring (breaking down a large integer into its prime factors in $\mathcal{O}(n^3)$) and discrete logarithms (finding the exponent in the expression $b^e = g(mod\ p)$, given b, g, and p, also in $\mathcal{O}(n^3)$) raised concerns about the security (more information about Shor's algorithm here: [21] and [22]). Consequently, security controls vulnerable to quantum attacks include cryptosystems built on Integer Factoring and Discrete Logarithms, such as **RSA, DSA, DH, ECDH, ECDSA, and their variants.** These systems are widely used in current security protocols. While symmetric key algorithms like **AES** are vulnerable to quantum attacks through **Grover's algorithm**, their security can be improved by doubling the key length. For instance, AES-128 is as resistant to classical computers as AES-256 would be to quantum computers, due to the limited advantages quantum computers possess in performing brute-force searches. And some symmetric-key cryptography forms, like **Vernam's One Time Pad**, guarantee perfect security against eavesdroppers. **Wegman-Carter Authentication** is also resistant to quantum attacks (more information here: [1]). Likewise, robust **hash functions** are expected to resist quantum adversaries. However, securely distributing symmetric keys remains a challenge, although options like **Quantum Key Distribution** exist for key establishment (more information here: [19]).

Potential frameworks for post-quantum cryptography focused on public-key algorithms are lattice-based, code-based, multivariate quadratic polynomials, as well as others.

2.4.3.1 Lattice-based cryptography

As explained in the paper [5] written by Daniel Bernstein, Johannes Buchmann and Erik Dahmen, **Lattice-based cryptographic constructions** exhibit significant promise in the realm of post-quantum cryptography due to their robust security proofs grounded in worst-case hardness, relatively **efficient implementations**, and inherent **simplicity**. These constructions rely on the presumed computational difficulty of lattice problems, comprising well-known problems such as:

- **Shortest Vector Problem (SVP):** Given a lattice basis B, find the shortest nonzero vector in $\mathcal{L}(\mathcal{B})$.
- **Closest Vector Problem (CVP):** Given a lattice basis B and a target vector t (not necessarily in the lattice), determine the lattice point $v \in \mathcal{L}(\mathcal{B})$ closest to t.
- **Shortest Independent Vectors Problem (SIVP):** Given a lattice basis $B \in \mathbb{Z}^{n \times n}$, discover n linearly independent lattice vectors $S = [s_1, \ldots, s_n]$ (where $s_i \in \mathcal{L}(\mathcal{B})$ for all i) minimizing the quantity $\|S\| = \max_i \|s_i\|$. This norm can be defined variably, but the Euclidean norm $\|x\| = \sqrt{\sum_i x_i^2}$ is the most commonly employed.

The most extensively studied algorithm for lattice problems is the **LLL algorithm**, formulated in 1982 by Lenstra, Lenstra and Lovász. It is a polynomial-time algorithm for SVP (and other fundamental lattice problems) that achieves an approximation factor of $2^{O(n)}$, where n represents

the lattice dimension. Despite its seemingly suboptimal approximation, the LLL algorithm finds versatile utility, ranging from factoring polynomials over rational numbers to integer programming and cryptanalysis (e.g., attacks on knapsack-based cryptosystems and specific instances of RSA).

Lattice-based cryptography systems are considered quantum-resistant because the presumed computational difficulty of lattice problems lacks a solution known through prime factorization or discrete logarithm, which are the only problems susceptible to rapid resolution through quantum parallelism so far.

2.4.3.2 Code-based cryptography

As explained in the paper "Code-based cryptography" from the book [5], **Code-based cryptography** relies on **error-correcting codes to fortify security measures**. By adding **redundancy** into digital data, these codes enable the **detection and rectification of errors** that might arise during transmission or storage. The foundation of code-based cryptography lies in the computational complexity of specific mathematical challenges associated with error-correcting codes, such as decoding a received message without knowledge of the decoding algorithm.

Goppa codes are an excellent illustration of effective error correction, especially when used as a secure coding system. This adaptation involves safeguarding the encoding and decoding functions while publicly sharing a disguised encoding function, allowing plaintext messages to be converted into scrambled code words. The recovery of plaintext is only possible with possession of the secret decoding function. The security of this approach hinges on the computational challenge of reversing it, which proves challenging for both conventional and quantum computers. It is based on syndrome decoding, a mathematical problem recognized as NP-complete in scenarios with unbounded errors, this technique emphasizes the robustness of code-based cryptography.

The **McEliece cryptosystem**, introduced by Robert McEliece in 1978, stands out as one of the earliest and most prominent examples of code-based cryptography. Founded on Goppa codes, this system relies on the challenging task of decoding these codes without access to specific secret parameters. Despite its resilience, the McEliece cryptosystem faces obstacles due to its requirement for **large key sizes**. Attempts to mitigate this issue through structured code variants have yielded successful attacks on some proposals. While code-based cryptography has seen proposals for signature schemes, its primary success lies in encryption schemes.

Code-based cryptography presents a promising avenue for post-quantum cryptographic research, due to involving fundamentally hard mathematical problems. However, practical implementations face challenges such as **large key sizes**, thereby trailing behind other quantum-safe alternatives.

2.4.3.3 Multivariate polynomial cryptography

According to the paper "Multivariate Public Key Cryptography" from the book [5], **Multivariate cryptosystems** are based on public-key systems and can be utilized for digital signatures. Multivariate (Public-Key) Cryptography involves the exploration of PKCs where the trapdoor one-way function - the secret key that allows the decryption of messages encrypted - is represented by a multivariate quadratic polynomial map over a finite field. Therefore, these schemes rely on solving systems of multivariate polynomials over finite fields. The public key typically comprises a **set of quadratic polynomials**:

$$\mathcal{P} = (p_1(w_1, ..., w_n), ..., p_m(w_1, ..., w_n))$$

Where each p_i is usually a quadratic nonlinear polynomial in $w = (w_1, ..., w_n)$:

$$z_k = p_k(w) := \sum_i P_{ik} w_i + \sum_i Q_{ik} w_i^2 + \sum_{i>j} R_{ijk} w_i w_j$$

And all coefficients $(P_{ik}, Q_{ik}, R_{ijk})$ and variables (w_i) are within $\mathbb{K} = \mathbb{F}_q$, the field with q elements. It is possible to determine the encryption or verification process by evaluating these polynomials at any given value. These systems are known as **multivariate public key cryptosystems (MPKCs)**. An equation system over a finite field, or **the Problem MQ**, must be solved to **invert a multivariate quadratic map**. The problem can be seen as:

$$\text{Solve the system } p_1(x) = p_2(x) = \ldots = p_m(x) = 0$$

Where each p_i is a quadratic polynomial in $x = (x_1, \ldots, x_n)$. The problem of solving such systems, known as the **MQ problem**, is **NP-hard**. However, a random set of quadratic equations does not always have the mathematical properties necessary to include a trapdoor (the secret key that allows the decryption of messages encrypted using the corresponding public key), and thus cannot be utilized in an MPKC.

The **Simple Matrix (or ABC)** encryption system is currently the most promising multivariate encryption scheme. This approach is highly efficient since it performs all computations in a single finite field and only solves linear systems during the decryption phase. There are other multivariate encryption systems, including **PMI and ipHFE**. However, these systems are typically inefficient because guesswork is involved in the decryption process. Although multivariate encryption systems have been proposed, multivariate cryptography has typically performed better when used for signatures. Prominent signature schemes include **UOV and Rainbow**.

2.4.3.4 Hash-based signatures

As explained in "Hash-based Digital Signature Schemes" from the book [5], **Hash-based cryptography** provides one-time signature schemes based on hash functions. While their resistance to quantum computers lacks definitive proof, these schemes have minimal security requirements. Furthermore, a new hash-based signature mechanism is introduced with every new cryptographic hash function. Therefore, constructs from symmetric cryptography are adequate for the building of safe signature schemes; sophisticated algorithmic difficulties in number theory or algebra are not needed. The underlying hash function can be selected based on the **available hardware and software resources**, which is a big advantage. An AES-based hash function, for instance, can be used to optimize runtime and reduce the code size of the signature scheme if it is meant to be implemented on a device that already supports AES. Consequently, if a vulnerability is detected in a secure hashing function, transitioning to a new and secure hash function is straightforward, ensuring continued effectiveness. Nevertheless, a significant disadvantage of Merkle-related schemes is their statefulness, which can cause problems in large-scale systems as the signer must keep track of which one-time signature keys have been used.

Hash-based signature schemes were introduced by **Ralph Merkle**. Merkle initially worked on one-time signature schemes, notably inspired by **Lamport and Diffie's scheme**. One-time signatures are foundational, requiring only a **one-way function** for construction. They do, however, have a serious drawback: a single key pair, consisting of a public verification key and a secret signature key, may only be used to sign and **verify a single document**, which is insufficient for the majority of applications. Using a **hash tree**, Merkle was the first to combine the validity of several one-time verification keys (the leaves of the hash tree) into a single public key (the hash tree's root). Each point on the binary tree is calculated as the hash of the concatenation of its child nodes, which is the basic idea behind the use of binary trees in hash signature methods.

Although Merkle's initial construction was not particularly efficient compared to the RSA signature scheme, numerous enhancements have since been made. For equivalent bit security, **XMSS** instantiated with AES-128 yields signatures that are more than ten times larger than those produced by RSA-2048. XMSS is a contemporary scheme currently **undergoing standardization**, built upon Merkle Trees with significant improvements, including enhanced efficiency in tree traversal and reduced private key sizes, as well as forward secrecy through the use of a pseudo-random number generator for creating one-time signature keys. Currently, hash-based signatures represent the most promising alternative to RSA and elliptic curve signature schemes.

2.4.4 Shor's algorithm

Shor's algorithm provides an efficient bounded error quantum polynomial-time algorithm that operates on a quantum computer. It can be used to break RSA or algorithms based on prime factorization. The goal is to find the factorization of a number N.

Firstly, we start by guessing a prime divisor of N, denoted as g. Since it's unlikely to be a divisor, we transform this guess into a much better one, $g^{(P/2)} \pm 1$. This is because there exists a P such that $g^P = m \cdot N + 1$. Thus, if we have $g^P = m \cdot N + 1$, then $g^P - 1 = m \cdot N$ and $(g^{(P/2)} + 1)(g^{(P/2)} - 1) = m \cdot N$.

The reasons why we cannot solve this system on classical computers are:

1. Even if we find $(g^{(p/2)}+1)(g^{(P/2)}-1) = m \cdot N$, $g^{(p/2)}-1$ may be a factor of m and $g^{(P/2)}+1$ may be a multiple of N and vice versa.
2. p may be odd, so $p/2$ is not an integer, leading to irrational numbers. The probability of p being even and the first setback not occurring is 37.5% of the time.
3. It's very time-consuming by trial and error, to find p, worse than a brute-force attack to divide N by all primes.

However, unlike classical computers, quantum computing allows parallelism and the ability to solve functions with different variables simultaneously, returning only one of the results. The key behind fast quantum computation is to set up a quantum superposition that calculates all possible answers at once while being cleverly arranged so that all of the wrong answers destructively interfere with each other.

The steps for Shor's algorithm are as follows: first, we randomly choose g and calculate the remainders of g raised to various powers modulo N. We want the remainder of this division to be 1, to have $g^p - 1$. For this, we know that:

$$g^x = m \cdot N + r \Rightarrow g^{(x+p)} = m_2 \cdot N + r,$$

because

$$g^x = m_x \cdot N + r,\ g^p = m_p \cdot N + 1,$$

so

$$(g^x)(g^p) = (m_x \cdot N + r)(m_p \cdot N + 1) = m_x \cdot m_p \cdot N^2 + m_x \cdot N + m_p \cdot N \cdot r + r = m_3 \cdot N + r$$

.

Hence, g^x, $g^{(x+p)}$, $g^{(x-p)}$, $g^{(x+2p)}$ all have the same remainder r modulo N. Thus, the goal is to find repeating remainders and measure their frequency to find p, through the Fourier transform.

For our superposition, we may apply a quantum version of the Fourier transform that repeats at a frequency of $1/p$. This would cause all potential erroneous frequencies to interact destructively, leaving us with only one quantum state: the number $1/p$.

The quantum Fourier transform yields a superposition of all other numbers when we input a single integer, but not just any old superposition. A superposition where the frequency of the single number we enter is roughly matched by the weights of the other numbers, which are all weighted differently. A greater number results in a superposition of all other numbers in the manner of a sine wave, but at a higher frequency.

Moreover, the sine waves add up or subtract and cancel out as we enter a superposition of numbers and output a superposition of superpositions. When we enter a superposition of integers separated by p, the sine waves interfere with one another, resulting in the single quantum state that represents $1/p$.

Chapter 3

Two quaternion based cryptosystem (Classical vs. Post-Quantum Protocols)

In this section, we will study several protocols, starting with their existing versions for classical computers and then examining adaptations found for post-quantum cryptography.

3.1 Diffie-Hellman protocol

According to [2], developed in 1976 by Whitfield Diffie and Martin Hellman, the Diffie-Hellman key exchange mechanism **allows two parties to send safe keys via an unprotected channel** without actually exchanging the keys. Each participant generates a matching public key and private key. The public keys are openly exchanged, leading to the establishment of a shared secret key for secure communication.

Widely adopted in security protocols like **TLS, SSH, and IPsec** for key generation, this method does face **vulnerabilities**, including a **lack of authentication** (which refers to validating the credentials presented by an entity seeking access) and susceptibility to attacks like **logjam**, potentially compromising security if not implemented correctly. **Logjam** attacks refer to the vulnerability within the TLS protocol, **downgrading connections to weaker cryptography levels**, allowing unauthorized access to the transmitted data (see [2] for further details).

To mitigate these risks, it is recommended to pair Diffie-Hellman with established authentication methods like **digital signatures to validate user identities** across public communication channels.

To set up the Diffie-Hellman scheme, two users collectively select positive whole numbers p and g, where, p represents a prime number, and g is a generator of $\mathbb{Z}_p^*$, the multiplicative group of integers modulo p, that are coprime to p . The generator g, also known as a primitive root

modulo p, possesses the unique property that, when raised to positive whole-number powers below $p-1$, it yields distinct results for any two such whole numbers. We choose p relatively large and in a way that g is relatively small.

After the receiver and the sender have privately settled on values for p and g, they select whole-number personal keys, denoted as a and b. These keys are smaller than $p-1$. Neither party shares their respective personal key with anyone.

Subsequently, the two users **calculate public keys denoted as A and B** from their respective personal keys using the following formulas:

$$A \equiv g^a \pmod{p}, \tag{3.1}$$

$$B \equiv g^b \pmod{p}. \tag{3.2}$$

The two parties then **transmit their respective public keys**, denoted as A and B, over an **unsecured communication channel**. Using these public keys, **each user can compute a resulting number,** x, derived from their own private keys, by raising the other user's public key to their own private key, as follows:

$$x_A \equiv A^b \equiv (g^a)^b \equiv (g^b)^a \equiv B^a \equiv x_B \pmod{p}. \tag{3.3}$$

In this way, both parties obtain the **same number** $x_A \equiv x_B$. Given that with high probability x is a large, seemingly random number, it is highly improbable for a potential hacker to accurately guess x, even with a powerful computer conducting numerous attempts. This allows the two users to theoretically communicate privately over a public channel, employing their chosen encryption method using the decryption key x.

The security of this scheme relies on the difficulty of solving the so-called discrete logarithm problem, that is defined as: given a prime number p, a generator g of the multiplicative group $\mathbb{Z}_p^*$, and an element $h \in \mathbb{Z}_p^*$, find an integer x such that $g^x \equiv h \pmod{p}$. In this case, given p, A and B find a and b. This can be easily solved with quantum computers as we have seen earlier.

3.2 Diffie-Hellman protocol based on Quaternions

Since classic commutative cryptography, such as the classical Diffie-Hellman protocol, is vulnerable to quantum computing, Mariana Durcheva and Kristian Karailiev proposed a new scheme in [**?**]. It leverages on the non-commutativity of quaternions (see section 2.3). Therefore, an adaptation of the Diffie-Hellman protocol is made, but based on quaternions.

Initially, both parties select three non-zero quaternions labeled as M, L, and S, which are

presumed to be public. Each user selects two polynomials with real coefficients as their secret keys. Let $p(x)$ and $t(x)$, be the secret keys polynomials of the first party. He computes his public key as

$$A = p(M) \cdot L \cdot t(S),$$

and shares it with the other party through a public channel. The other party selects two polynomials with real coefficients, $q(x)$ and $r(x)$, as secret keys, and computes his public key

$$B = q(M) \cdot L \cdot r(S),$$

which is transmitted to the first party via a public channel.

Both parties calculate their secret keys, k_A and k_B, respectively, as:

$$k_A = p(M) \cdot B \cdot t(S) = p(M) \cdot q(M) \cdot L \cdot r(S) \cdot t(S), \tag{3.4}$$

$$k_B = q(M) \cdot A \cdot r(S) = q(M) \cdot p(M) \cdot L \cdot t(S) \cdot r(S). \tag{3.5}$$

We know that the keys are the same because, for a given quaternion H and two polynomials with real coefficients $P_1(x)$ and $P_2(x)$, we have:

$$P_1(H) \cdot P_2(H) = P_2(H) \cdot P_1(H). \tag{3.6}$$

This holds since, real numbers commute with quaternions, and any two powers of the same quaternion also commute.

Thus, in analogy to the Diffie-Hellman protocol, a key is computed using the private and public keys unique to each user, and the final key is known solely by those users.

This paper is an adaptation of the previous paper "A Diffie-Hellman compact model over non-commutative rings using quaternions" [13], authored by J. Kamlofsky, J. P. Hecht, Izzi, and Masuh. In this paper a slightly different scheme is proposed. However, due to inconsistencies in the example provided, it is challenging to understand.

The adaptations of Diffie-Hellman presented here pose an equal challenge for a classical attacker or one with the capabilities of a quantum computer. This is because not only post-quantum algorithms for integer factorization and discrete logarithm do not apply, but there is also no documentation on the complexity of problems using quaternions in the Complexity Zoo site, that has most of the complexity problems documented ([15]). Nevertheless, in this case, the difficulty lies in the fact that even if the attacker knows $M, L, S, A,$ and B, it seems hard to discover the polynomials, based solely on this information, $p(x), t(x), q(x)$, and $r(x)$ that satisfy (3.4) and (3.5).

3.3 NTRU - Public key Encryption Protocol

In their paper titled "QTRU: Quaternionic Version of the NTRU Public-Key Cryptosystems" [18], Malekian, Zakerolhosseini and Mashatan described the following protocol, which uses polynomial multiplication and modular algebra.

Firstly, is necessary to choose an integer $n > 1$, and two coprime numbers q and p such that $q >> p$. The domain of the polynomials used is $\mathcal{R} = \mathbb{Z}[x]/\langle x^n - 1\rangle$, the ring of polynomials with integer coefficients modulo $x^n - 1$. In this ring, one can always choose a representation for the polynomials of degree at most $n - 1$. Furthermore, we consider $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n - 1\rangle$ and $\mathcal{R}_p = \mathbb{Z}_p[x]/\langle x^n - 1\rangle$, where the polynomial coefficients are taken modulo q and p, respectively.

We define the subsets $\mathcal{L}_d$, $\mathcal{F}_d$ and $\mathcal{M}$ of $\mathbb{Z}[x]/\langle x^n - 1\rangle$ in the following way: $\mathcal{L}_d$ and $\mathcal{F}_d$ contain polynomials with coefficients from the set $\{-1, 0, 1\}$ in $\mathbb{Z}[x]/\langle x^n - 1\rangle$, where in $\mathcal{L}_d$, d is the number of monomials with coefficients equal to one and the same number of minus ones, while in $\mathcal{F}_d$ polynomials have d ones and $(d - 1)$ minus ones. The set $\mathcal{M}$ comprises the polynomials in $\mathbb{Z}[x]/\langle x^n - 1\rangle$ with coefficients ranging from $-\frac{p-1}{2}$ to $\frac{p-1}{2}$.

To expound the NTRU protocol, this cryptographic scheme will be presented alongside an example that runs throughout the system description. We choose the following parameters: $n = 17, p = 3, q = 127, d = 5$, and the message used for this example is "NTRU". For more details about the code, please see the section A.1 in the appendix.

3.3.1 Key generation

Parties begin by selecting an integer $n > 1$, and two coprime numbers q and p such that $q >> p$. Typically, n is prime, and p is 3. These numbers are public. Then one must also select d_f, d_g and d_r, such that these numbers are less then $n/2$.

The first party starts by generating two random polynomials $F \in \mathcal{F}_{d_f}$, and $G \in \mathcal{L}_{d_g}$, and such that F must be invertible both in $\mathcal{R}_q$ and in $\mathcal{R}_p$, with the respective inverses being denoted by F_q and F_p. In other words, $FF_p \equiv 1 \pmod p$ and $FF_q \equiv 1 \pmod q$.

```
F: -x^23 + x^16 - x^15 + x^13 - x^11 + x^9 + x^8 + x^2 - x
```

```
F_p:  x^26 + x^24 + x^23 + 2*x^21 + x^19 + x^16 + 2*x^14 + 2*x^13 + 2*x
   ^9 + x^8 + 2*x^7 + 2*x^6 + x^5 + x^2 + 2
```

```
F_q: 77*x^26 + 72*x^25 + 93*x^24 + 99*x^23 + 126*x^22 + 96*x^21 + 71*x
    ^20 + 59*x^19 + 24*x^18 + 37*x^17 + 98*x^16 + 53*x^15 + 110*x^14 +
    47*x^13 + 51*x^12 + 98*x^11 + 21*x^10 + 79*x^9 + 20*x^8 + 33*x^7 +
    124*x^6 + 111*x^5 + 9*x^4 + 126*x^3 + 13*x^2 + 43*x + 116
```

```
G: x^25 + x^22 - x^20 + x^19 - x^15 - x^13 - x^9 + x^3 - x^2 + x
```

The private keys of the first party are thus: F and F_p. The public key, denoted H, is calculated as follows:

$$H = F_q G \pmod{q} \tag{3.7}$$

```
H = F_q*G (mod q) : 111*x^26 + 78*x^25 + 31*x^24 + 75*x^23 + 120*x^22 +
    114*x^21 + 82*x^20 + 32*x^19 + 105*x^18 + 23*x^17 + 29*x^16 + 69*x
    ^15 + 50*x^14 + 6*x^13 + 19*x^12 + 85*x^11 + 48*x^10 + 74*x^9 + 113*
    x^8 + 122*x^7 + 94*x^6 + 5*x^5 + 64*x^4 + 108*x^3 + 90*x^2 + 7*x +
    24
```

3.3.2 Encryption

For the second party to encrypt a message with the public key H, the message must first be represented as a polynomial $M \in \mathcal{M}$. Next, a random polynomial $R \in \mathcal{L}_d$ is generated, with $d = d_r$. Details on how to encode and decode these strings to and from the format required can be found in section A.1 in the appendix.

```
Text: NTRU
M: -x^24 - x^23 - x^22 + x^19 + x^16 + x^13 + x^9 + x^7 + x^4 + x^3
```

```
R: -x^26 - x^23 + x^21 + x^20 - x^18 - x^16 + x^12 - x^9 + x^7 + x^4
```

To calculate the encrypted message, it performs:

$$E = pHR + M \pmod{q} \tag{3.8}$$

```
E = p*H*R+M (mod q): 82*x^26 + 112*x^25 + 58*x^24 + 81*x^23 + 33*x^22 +
    52*x^21 + 78*x^20 + 111*x^19 + 121*x^18 + 34*x^17 + 70*x^16 + 94*x
    ^15 + 118*x^14 + 4*x^13 + 95*x^12 + 8*x^11 + 5*x^10 + 5*x^9 + 18*x^8
     + 76*x^7 + 14*x^6 + 101*x^5 + 103*x^4 + 6*x^3 + 99*x^2 + 10*x + 67
```

We cannot determine M by computing $E \pmod p$ because the modulo operation $\mod q$ impacts the entire expression $pHR + M$. Furthermore, $\mod p$ and $\mod q$ typically do not yield the same result (i.e., $a \pmod p$ and $a \pmod q$ generally produce different outcomes).

In the example presented we can see that:

$$\begin{aligned} E &= 82 * x^{26} + 112 * x^{25} + 58 * x^{24} + 81 * x^{23} + 33 * x^{22} + 52 * x^{21} + 78 * x^{20} + 111 * x^{19} \\ &+ 121 * x^{18} + 34 * x^{17} + 70 * x^{16} + 94 * x^{15} + 118 * x^{14} + 4 * x^{13} + 95 * x^{12} + 8 * x^{11} \\ &+ 5 * x^{10} + 5 * x^9 + 18 * x^8 + 76 * x^7 + 14 * x^6 + 101 * x^5 + 103 * x^4 + 6 * x^3 + 99 * x^2 \\ &+ 10 * x + 67 \pmod q \\ &\equiv x^{26} + x^{25} + x^{24} + x^{21} + x^{18} + x^{17} + x^{16} + x^{15} + x^{14} + x^{13} + 2 * x^{12} + 2 * x^{11} \\ &+ 2 * x^{10} + 2 * x^9 + x^7 + 2 * x^6 + 2 * x^5 + x^4 + 1 * x + 1 \pmod p \end{aligned}$$

and:

$$\begin{aligned} M &= -x^{24} - x^{23} - x^{22} + x^{19} + x^{16} + x^{13} + x^9 + x^7 + x^4 + x^3 \\ &\equiv 2x^{24} + 2x^{23} + 2x^{22} + x^{19} + x^{16} + x^{13} + x^9 + x^7 + x^4 + x^3 \pmod p \end{aligned}$$

so $E \neq M \pmod p$.

3.3.3 Decryption

To decrypt E, the first party uses its private key F to compute:

$$B = FE \equiv F(pHR + M) \equiv pFF_qGR + FM \equiv pGR + FM \pmod q \tag{3.9}$$

```
B: 122*x^26 + 2*x^24 + 124*x^23 + 114*x^22 + 2*x^21 + 12*x^20 + x^19 +
   4*x^18 + 5*x^17 + 4*x^15 + 6*x^14 + 11*x^13 + 15*x^12 + 125*x^11 +
   121*x^10 + 12*x^9 + 125*x^8 + 125*x^7 + 121*x^6 + 125*x^5 + 119*x^4
   + 115*x^3 + 122*x^2 + 5*x + 118
```

Then we adjust the coefficients of B to be in $[-\frac{q-1}{2}, \frac{q-1}{2}]$, denoted by B' the resulting polynomial, ensuring that pGR are equal to 0 modulo p. This adjustment is necessary because G and R have coefficients in $\{-1, 0, 1\}$, resulting in coefficients of $p \cdot G \cdot R$ being in $\{-p, 0, p\}$. However, when taken modulo q, these coefficients become $\{0, p, q-p\}$, and since $q-p$ modulo p is always different from $-p$ modulo p (as p cannot divide q), adjustments are needed.

However, the same adjustment cannot be applied to $x = p \cdot F_q \cdot G \cdot R$, as F_q has coefficients in $\mathbb{Z}_q$. Thus, modulo p, x becomes $x + kq$, which is not necessarily equal to x modulo p, unless k is a multiple of p.

Then we compute $A = B' \pmod p$ which eliminates the first term of (3.9):

$$A = B'(mod\ p) \equiv pGR + FM(mod\ p) \equiv FM(mod\ p) \tag{3.10}$$

```
B_adj: -5*x^26 + 2*x^24 - 3*x^23 - 13*x^22 + 2*x^21 + 12*x^20 + x^19 +
    4*x^18 + 5*x^17 + 4*x^15 + 6*x^14 + 11*x^13 + 15*x^12 - 2*x^11 - 6*x
    ^10 + 12*x^9 - 2*x^8 - 2*x^7 - 6*x^6 - 2*x^5 - 8*x^4 - 12*x^3 - 5*x
    ^2 + 5*x - 9
```

```
A = B_adj (mod p): x^26 + 2*x^24 + 2*x^22 + 2*x^21 + x^19 + x^18 + 2*x
    ^17 + x^15 + 2*x^13 - 2*x^11 - 2*x^8 - 2*x^7 - 2*x^5 + x^4 + x^2 +
    2*x
```

Finally we multiply F_p by $FM \pmod p$ to obtain M. And so we try to decode M to obtain the message, depending on the probability of a successful decryption.

```
M = F_p*F*M (mod p): 2*x^24 + 2*x^23 + 2*x^22 + x^19 + x^16 + x^13 + x^9
     + x^7 + x^4 + x^3
Reconstructed_text: NTRU
```

3.3.3.1 Successful decryption

The decryption is successful only if the coefficients of $X = pGR + FM$ in $\mathcal{R}$ are in the interval $[\frac{-q+1}{2}, \frac{q-1}{2}]$. If f_i are the components of F, g_i of G, r_i of R, m_i of M, and x_i of X, we have the following probabilities of each coefficient of these polynomials being 1, 0, or -1 :

$$P(f_i = 1) = \frac{d_f}{n}, \quad P(f_i = -1) = \frac{d_f - 1}{n} \approx \frac{d_f}{n}, \quad P(f_i = 0) = \frac{N - 2d_f}{n}$$

$$P(g_i = 1) = \frac{d_g}{n}, \quad P(g_i = -1) = \frac{d_g}{n}, \quad P(g_i = 0) = \frac{N - 2d_g}{n}$$
$$P(r_i = 1) = \frac{d_r}{n}, \quad P(r_i = -1) = \frac{d_r}{n}, \quad P(r_i = 0) = \frac{N - 2d_r}{n}$$
$$P(m_i = j) = \frac{1}{p}, \quad \text{for } j \in \left]-\frac{p-1}{2}, \frac{p-1}{2}\right].$$

The expected value given by $E[Y] = \sum_i y_i \cdot P(Y = y_i)$ is:

$$E(f_i) = E(g_i) = E(r_i) = \frac{d}{n} - \frac{d}{n} = 0$$

$$E(m_i) = \frac{1}{p} \cdot \sum_{i=\frac{-p+1}{2}}^{\frac{p-1}{2}} i = 0.$$

Thus, $E(x_i) = 0$.

On the other hand,

$$\text{Var}(f_i) = \frac{d_f}{n} \cdot (-1 - E(f_i))^2 + \frac{d_f}{n} \cdot (1 - E(f_i))^2 + \frac{d_f}{n} \cdot (0 - E(f_i))^2 = 2\frac{d_f}{n}.$$

Similarly, $\text{Var}(g_i) = 2\frac{d_g}{n}$ and $\text{Var}(r_i) = 2\frac{d_r}{n}$.

$$\text{Var}(m_i) = \frac{1}{p} \cdot \sum_{i=\frac{-p+1}{2}}^{\frac{p-1}{2}} i^2 = \frac{1}{p} \cdot 2 \cdot \sum_{i=0}^{(p-1)/2} i^2 = \frac{1}{p} \cdot 2 \cdot \frac{\left(\frac{p-1}{2}\right)\left(\frac{p-1}{2} - 1\right)\left(2\frac{p-1}{2} - 1\right)}{6} = \frac{(p-1)(p+1)}{12}.$$

Since the coefficients of the functions are independent variables, the covariance is always 0. Therefore, we have:

$$\text{Var}(x_i) = \text{Var}(p \cdot g_i * r_i + f_i * m_i) = p^2 \cdot \text{Var}(g_i * r_i) + \text{Var}(f_i * m_i) = p^2 \cdot 4\frac{d_g \cdot d_r}{N^2} + \frac{d_f \cdot (p-1) \cdot (p+1)}{6 \cdot N}.$$

From the expected value and variance of the coefficients of X, the probability of a successful decryption of a single coefficient is equivalent to:

$$P\left(\frac{-q+1}{2} \leq x_i \leq \frac{q-1}{2}\right) = 2 \cdot P\left(x_i \leq \frac{q-1}{2}\right) - 1 = 2 \cdot \rho\left(\frac{q-1}{2\sigma}\right) - 1$$

where ρ is the standard normal distribution function and $\sigma = \sqrt{\text{Var}(x_i)}$.

So the probability of a successful decryption is:

$$P(\text{success decrypt}) = \left(2 \cdot \rho\left(\frac{q-1}{2\sigma}\right) - 1\right)^N$$

In the paper mentioned earlier [18], the authors claim that the decryption process will never fail if the public parameters (N, p, q, d) are chosen to satisfy $q > (6d + 1)p$ (where $d := d_f = d_g = d_r$,

as defined earlier). However, for improved performance and to reduce the size of the public key, the parameter q can be selected so that the probability of decryption failure is very small.

In the specific case illustrated, the probability of success is 1, because of what the authors claim in the paper. However, if the parameters were, for example: $N = 17, p = 3, q = 11, d = 5$, the probability of successful decryption would be ≈ 0.88.

According to [6], compared to well-known systems like RSA or ECC, NTRU's primary advantage lies in its computational efficiency, performing basic arithmetic operations with low inherent complexity, approximately $O(n^2)$ in worst-case scenarios where n is relatively small. Additionally, since NTRU does not rely on integer factorization or discrete logarithms, it is not susceptible to quantum attacks, differing from classical systems as understood to date.

The security of NTRU is based on the difficulty of finding short vectors in Convolution Modular Lattices (CML), with lattice-reduction techniques playing a crucial role. Although potential attacks using lattice-reduction algorithms like LLL can solve the Shortest Vector Problem (SVP) to within a factor of $2^{O(n)}$ for suitable n dimensions, NTRU remains secure with sufficiently large n values, ensuring practical security levels, as referred in [6].

3.4 QTRU - Public key Encription Protocol

According to the paper [18], mentioned in the previous section, QTRU is an approach based on NTRU but using quaternions. The quaternion algebra used is $\left(\frac{-1,-1}{\mathbb{F}}\right)$, where $\mathbb{F}$ is a certain field. Then, we choose n, p, q as primes with $p \ll q$, allowing us to define $\mathbb{A} = \left(\frac{-1,-1}{\mathbb{Z}/<x^n-1>}\right)$, $\mathbb{A}_p = \left(\frac{-1,-1}{\mathbb{Z}_p/<x^n-1>}\right)$, and $\mathbb{A}_q = \left(\frac{-1,-1}{\mathbb{Z}_q/<x^n-1>}\right)$. Moreover we need to define the parameters d_f, d_g and d_r.

As in NTRU, we define the subsets $\mathcal{L}_d$, $\mathcal{F}_d$ and $\mathcal{M}$ of $\mathbb{Z}[x]/\langle x^n - 1\rangle$. In the following way: $\mathcal{L}_d$ and $\mathcal{F}_d$ contain polynomials with coefficients from the set $\{-1, 0, 1\}$ in $\mathbb{Z}[x]/\langle x^n - 1\rangle$, where in $\mathcal{L}_d$, d is the number of monomials with coefficients ones and the same number of minus ones, while in $\mathcal{F}_d$, polynomials have d ones and $(d-1)$ minus ones. The set $\mathcal{M}$ comprises the polynomials in $\mathbb{Z}[x]/\langle x^n - 1\rangle$ with coefficients ranging from $-\frac{p-1}{2}$ to $\frac{p-1}{2}$.

For a more practical approach, to understand the QTRU protocol, an example will be showcased alongside the proposed scheme explanation. The defined parameters are: $n = 7,\ q = 127,\ p = 3,\ d_f = d_g = d_r = 2$, and the message used for this example is: "QTRU". To see more details about the code used please go to section A.2 in the appendix.

3.4.1 Key generation

To initiate the key generation, the first party selects $F = f_0 + f_1 i + f_2 j + f_3 k$, where $f_0, f_1, f_2, f_3 \in \mathcal{F}_d$, where $d = d_f$. This polynomial F must be invertible in both $\mathbb{A}_p$ and $\mathbb{A}_q$, which only requires its norm to be invertible in $\mathbb{Z}_p/ < x^n - 1 >$ and $\mathbb{Z}_q/ < x^n - 1 >$.

Next, the inverse of F in $\mathbb{A}_p$ and $\mathbb{A}_q$ is calculated as:

$$F_p^{-1} \equiv \frac{\bar{F}}{N(F)} \pmod{p}$$

$$F_q^{-1} \equiv \frac{\bar{F}}{N(F)} \pmod{q}$$

```
F: (x^6 + x^2 - 1) + (x^6 + x^5 - x)i + (-x^6 + x^3 + x^2)j + (-x^3 + x
    + 1)k
```

```
F_p: (2*x^6 + 2*x^5 + 2*x^4 + 2*x^3 + 2*x^2 + 2*x + 1) + (x^4 + 2*x^3 +
    2*x^2 + 2*x + 1)i + (x^6 + 2*x^4 + 2)j + (x^6 + 2*x^5 + 2*x^4 + 2*x
    ^3 + x^2)k
```

```
F_q: (47*x^6 + 92*x^5 + 117*x^4 + 105*x^3 + 103*x^2 + 69*x + 7) + (16*x
    ^6 + 110*x^5 + 116*x^4 + 30*x^3 + 99*x^2 + 62*x + 43)i + (72*x^6 +
    18*x^5 + 35*x^4 + 39*x^3 + 64*x^2 + 45*x + 76)j + (116*x^6 + 30*x^5
    + 99*x^4 + 62*x^3 + 43*x^2 + 16*x + 110)k
```

Then, the first party creates $G = g_0 + g_1 i + g_2 j + g_3 k$, where $g_0, g_1, g_2, g_3 \in \mathcal{L}_d$, and $d = d_g$.

```
G: (-x^6 + x^3 + x - 1) + (x^6 - x^5 - x^4 + 1)i + (-x^6 + x^3 + x - 1)j
     + (-x^4 + x^2 + x - 1)k
```

The public key H is then computed as $H = F_q^{-1} G \pmod q$ and transmitted to the second party through an insecure channel. The private keys are F and F_p^{-1}.

```
H = F_q*G: (112*x^6 + 63*x^5 + 125*x^4 + 48*x^3 + 65*x^2 + 70*x + 25) +
    (14*x^6 + 40*x^5 + 191*x^4 + 85*x^3 + 87*x^2 + 183*x + 162)i + (108*
    x^6 + 136*x^5 + 183*x^4 + 108*x^3 + 129*x^2 + 56*x + 169)j + (94*x^6
     + 50*x^5 + 121*x^4 + 151*x^3 + 188*x^2 + 160*x + 125)k
```

3.4.2 Encryption

To use the public key for encryption, the second party begins by creating a random polynomial $R = r_0 + r_1 i + r_2 j + r_3 k$, where $r_0, r_1, r_2, r_3 \in \mathcal{L}_d$, where $d = d_r$. The message is encoded into a polynomial $M = m_0 + m_1 i + m_2 j + m_3 k$, where $m_0, m_1, m_2, m_3 \in \mathcal{M}$, with coefficients within $-\frac{p-1}{2}$ and $\frac{p-1}{2}$, and so the message is encoded in base p. Details on how to encode and decode these strings to and from the format required can be found in section A.2 in the appendix.

```
R: (x^5 - x^4 + x^2 - x) + (x^6 + x^4 - x - 1)i + (x^6 - x^3 - x^2 + x)j
     + (x^5 + x^4 - x^2 - x)k
```

```
Text: QTRU
M:  (x^5) + (x^6 + x^3 + 1)i + (x^3 + x)j + (x^5 + x^4)k
```

Then, the second party calculates $E = pHR + M \pmod q$ and transmits the encrypted message to the first party.

```
E = p*H*R+M (mod q): (59*x^6 + 44*x^5 + 82*x^4 + 120*x^3 + 24*x^2 + 37*x
    + 16) + (122*x^6 + 29*x^5 + 122*x^4 + 82*x^3 + 70*x^2 + 74*x + 12)i
    + (57*x^6 + 20*x^5 + 13*x^4 + 46*x^3 + 70*x^2 + 126*x + 51)j + (94*
   x^6 + 62*x^5 + 122*x^4 + 49*x^3 + 105*x^2 + 48*x + 30)k
```

3.4.3 Decryption

The first party decrypts E as follows:

$$\begin{aligned} B &= FE \pmod q \\ &\equiv FpHR + F * M \pmod q \\ &\equiv pFF_q^{-1}GR + FM \pmod q \\ &\equiv pGR + FM \pmod q \end{aligned}$$

```
B = F*E (mod q): (121*x^6 + 112*x^5 + 126*x^4 + 11*x^3 + 115*x^2 + 10*x
   + 7) + (4*x^6 + 11*x^5 + 118*x^4 + 107*x^3 + 125*x^2 + 13*x + 7)i +
   (119*x^6 + 120*x^5 + 117*x^4 + 125*x^3 + 8*x^2 + 13*x + 10)j + (6*x
   ^6 + 121*x^5 + 126*x^4 + 121*x^3 + 111*x^2 + 17*x + 8)k
```

We must adjust the coefficients of the four polynomials in B to B', so that they fall within the interval $\left[-\frac{q-1}{2}, \frac{q-1}{2}\right]$, ensuring that pGR vanishes modulo p. This adjustment is necessary because G and R have coefficients in $\{-1, 0, 1\}$, resulting in coefficients of $p \cdot G \cdot R$ being in $\{-p, 0, p\}$. However, when taken modulo q, these coefficients become $\{0, p, q-p\}$, and since $q - p$ modulo p is always different from $-p$ modulo p (as p cannot divide q), adjustments are needed.

However, the same adjustment cannot be applied to $x = p \cdot F_q^{-1} \cdot G \cdot R$, as F_q^{-1} has coefficients in $\mathbb{Z}_q$. Thus, modulo p, x becomes $x + kq$, which is not necessarily equal to x modulo p, unless k is a multiple of p.

$$\begin{aligned} A &= B' \pmod p \\ &\equiv pGR + FM \pmod p \\ &\equiv FM \pmod p \end{aligned}$$

$$M' = F_p^{-1}A \equiv F_p^{-1}FM \pmod{p}$$

```
B_adj: (-6*x^6 - 15*x^5 - x^4 + 11*x^3 - 12*x^2 + 10*x + 7) + (4*x^6 +
    11*x^5 - 9*x^4 - 20*x^3 - 2*x^2 + 13*x + 7)i + (-8*x^6 - 7*x^5 - 10*
    x^4 - 2*x^3 + 8*x^2 + 13*x + 10)j + (6*x^6 - 6*x^5 - x^4 - 6*x^3 -
    16*x^2 + 17*x + 8)k
```

```
A = B_adj (mod p): (2*x^4 + 2*x^3 + x + 1) + (x^6 + 2*x^5 + x^3 + x^2 +
    x + 1)i + (x^6 + 2*x^5 + 2*x^4 + x^3 + 2*x^2 + x + 1)j + (2*x^4 + 2*
    x^2 + 2*x + 2)k
```

Finally, M' is adjusted so that its coefficients are within $-\frac{p-1}{2}$ and $\frac{p-1}{2}$, resulting in the encoding of M into a quaternion of polynomials. Reversing this operation retrieves the message if we have a successful decryption.

```
M_adj = F_p*A (mod p): (x^5) + (x^6 + x^3 + 1)i + (x^3 + x)j + (x^5 + x
    ^4)k
Reconstructed_text: QTRU
```

3.4.3.1 Successful decryption

The probability of a successful decryption, similar to NTRU, depends on the coefficients of polynomials in each component of the quaternion $X = FE = (pGR + FM)$ in $\mathbb{A}$, lying within $\left[\frac{-q+1}{2}, \frac{q-1}{2}\right]$.

In this case, if $X = x_0 + x_1 i + x_2 j + x_3 k$, then:

$$x_0 = pg_0r_0 - pg_1r_1 - pg_3r_3 - pg_2r_2 + f_0m_0 - f_1m_1 - f_3m_3 - f_2m_2$$

$$x_1 = pg_0r_1 + pg_1r_0 - pg_3r_2 + pg_2r_3 + f_0m_1 + f_1m_0 - f_3m_2 + f_2m_3$$

$$x_2 = pg_3r_1 + pg_2r_0 + pg_0r_2 - pg_1r_3 + f_3m_1 + f_2m_0 + f_0m_2 - f_1m_3$$

$$x_3 = pg_1r_2 + pg_0r_3 - pg_2r_1 + pg_3r_0 + f_1m_2 + f_0m_3 - f_2m_1 + f_3m_0$$

The expected value of every coefficient of all components of F, G, R, and M, as in NTRU, continues to be 0, so the expected value of each coefficient of all components of X is also 0. The variance of each coefficient of each component of F, G, R, and M is also the same as in

NTRU. However, the variance of each coefficient of each component of X is now different due to the calculations presented above for each component. Nevertheless, we can verify that

$$\begin{aligned}\mathsf{Var}(x_{0,k}) = \mathsf{Var}\Bigg(& \sum_{i+j=k(\mathsf{mod}N)} (pg_{0,i}r_{0,j} - pg_{1,i}r_{1,j} - pg_{3,i}r_{3,j} - pg_{2,i}r_{2,j} + \\ & f_{0,i}m_{0,j} - f_{1,i}m_{1,j} - f_{3,i}m_{3,j} - f_{2,i}m_{2,j})\Bigg) \\ & = N4\frac{4p^2 d_r d_g}{N^2} + N4\frac{d_f(p-1)(p+1)}{N*6} \\ & = \frac{16p^2 d_r d_g}{N} + \frac{4d_f(p-1)(p+1)}{6} \\ & = \mathsf{Var}(x_{1,k}) = \mathsf{Var}(x_{2,k}) = \mathsf{Var}(x_{3,k})\end{aligned}$$

The probability of decrypting a single coefficient of a component of X is given by:

$$P\left(\frac{-q+1}{2} \leq x_i \leq \frac{q-1}{2}\right) = 2 \cdot P\left(x_i \leq \frac{q-1}{2}\right) - 1 = 2 \cdot \rho\left(\frac{q-1}{2\sigma}\right) - 1$$

where ρ is the standard normal distribution function and $\sigma = \sqrt{\mathsf{Var}(x_{0,k})}$.

So, the probability of a successful decryption of only one component of X is:

$$P(\text{successful decryption of } x_0) = \left(2 \cdot \rho\left(\frac{q-1}{2\sigma}\right) - 1\right)^N$$

Finally, the probability of a successful decryption of X is:

$$P(\text{successful decryption of } x_0) = \left(2 \cdot \rho\left(\frac{q-1}{2\sigma}\right) - 1\right)^{(4N)}$$

In the example presented, the success probability was ≈ 1. However, if the parameters were different, such as $N = 17$, $p = 3$, $q = 127$, and $d = 8$, the likelihood of a successful decryption would be only around 0.53.

According to the paper studied [18], the authors conclude that QTRU is more secure than NTRU because its lattice structure does not completely fit into the category of Convolutional Modular Lattices (CML). Despite being four times slower than NTRU under the same conditions, QTRU offers greater resistance to lattice-based attacks. This increased security allows for compensating the speed loss by reducing the dimension while maintaining the same security level. Moreover, QTRU can be enhanced with parallel algorithms.

In summary, QTRU provides a more secure alternative to NTRU with the potential for optimization and adaptation to various algebraic structures for designing new lattice-based cryptosystems.

And, just as NTRU is resistant to both classical and quantum attacks, QTRU is resistant to attacks in both scenarios.

Chapter 4

Conclusions

With the advent of quantum computers and the subsequent development of post-quantum algorithms, several cryptographic methods currently in use have been shown to be vulnerable to quantum attacks. Notably, algorithms such as RSA, DSA, DH, ECDH, ECDSA, and their variants are susceptible because they rely on problems considered difficult for classical computers but solvable by quantum computers, specifically the discrete logarithm problem and prime factorization. To address these vulnerabilities, it is necessary to explore algorithms that do not fundamentally depend on these problems.

In this thesis, we studied and implemented cryptographic schemes that utilize quaternions, a non-commutative structure, which makes them resistant to quantum attacks. We began by investigating schemes that employ quaternions and selected two: an adaptation of the Diffie-Hellman protocol and an adaptation of the NTRU protocol.

Through this study, we confirmed that the traditional Diffie-Hellman protocol, based on the discrete logarithm problem, is vulnerable to quantum attacks. However, in the quaternion-based version of this scheme, no existing algorithm can efficiently solve the underlying problem, making it resistant to quantum attacks. Furthermore, NTRU is based on lattice problems such as the Shortest Vector Problem (SVP) and the Closest Vector Problem (CVP), which are NP-hard, and so is resistant to quantum attacks. Since QTRU is derived from NTRU and incorporates non-commutativity, it also exhibits quantum resistance.

In addition to reviewing these papers, we implemented the schemes to enhance understanding, as the use of advanced abstract algebra made the original papers challenging to comprehend. During our study, we encountered difficulties in understanding a cryptographic method called BQTRU, proposed by Bagheri, Khadijeh, Sadeghi, Mohammad-Reza, and Panario. This method introduces polynomials with two variables, adding a layer of complexity. And so, to continue this work, we propose further analysis and comprehension of this last paper, as it employs more advanced concepts and, according to the presented results, is an efficient algorithm.

In conclusion, to contribute to the advancement of these more secure cryptographic schemes

and aid the reader's understanding, I have studied and implemented various cryptographic methods using quaternions in this thesis. Due to their properties, these methods are resistant to attacks from both classical and quantum computers.

Appendix A

Code

To experiment the system studied, we used SageMath because most of the mathematical structures needed are already supported by their built-in packages. With these implementations, we were able to test NTRU and QTRU.

A.1 NTRU

As explained earlier, in this example of NTRU we start by defining $n = 127$, $p = 3$, $q = 127$, and also $d_f = 5$, $d_g = 5$ and $d_r = 5$ (A.1), as parameters to perform A.2. We also choose our text to be encrypted and the parameters to then code it into a list of polynomials modulo p (A.3).

```
1 ntru_system(27,127,3,5,5,5,"NTRU")
```

Listing A.1: NTRU - Initialization and Encryption proccess

Following the creation of the polynomials list, we generate a corresponding number of public keys from the first party. To prevent possible dictionary attacks, we choose to encrypt each polynomial with different key pairs. Then, the first party generates each of these public keys and the corresponding private key (A.4). This involves generating two random polynomials, F and G (A.6 and A.5, resp.), and computing the inverse of F modulo p and q.

For the second party to perform the encryption, for each polynomial from the message to encrypt, it generates a random polynomial R with d_r number of ones and minus ones (as performed in A.5). The list of encrypted polynomials is then given by $E = p \cdot (H \cdot R) + M \pmod{q}$.

Finally, the first party can decrypt the message (A.7), decode the polynomials back to the text (A.8), and recover the original message.

```
def ntru_system(n,q,p,d_f,d_g,d_r,text):
    pols = text_to_pol_ntru(text,p,n)  # calculated by party 2
    keys=len(pols) # given to party 1 by party 2 to calculate all key
       pairs
    publicKeys=[]
    privateKeys=[]
    for i in range(keys):
        H, f, f_p=publicKey_ntru(n,p,q,d_f,d_g)
        publicKeys.append(H) # given to party 2 by party 1
        privateKeys.append([f,f_p]) # kept a secret by party 1
    encrypt=[]
    for i in range(keys):
            M=pols[i]
            r=rand_poly_ntru(d_r,d_r,n)
            H=publicKeys[i]
            aux=p*(H*r)
            E=(aux+M).mod(q) # party 2 calculates the encryption and
               gives it to party 1
            encrypt.append(E)
    m=[]
    for i in range(keys):
        M=decrypt_ntru(privateKeys[i][0],privateKeys[i][1],encrypt[i],p,
           q,n)
        m.append(M) # party 1 decrypts the message with its private keys
    reconstructed_text = pol_to_text_ntru(m,p,n) # party 1 decodes the
       message and recover the message
```

Listing A.2: NTRU - Initialization and Encryption proccess

Firstly, every time we create a polynomial in a function we need to define *R.<x>* so that Sagemath defines x as a generator and the coefficients as integers. Using the *text_to_pol_ntru()* function, we code our message letter by letter transforming its ASCII code to base p with *text_to_base_p()*, adjust the values to be between $-p/2+1$ and $p/2$ with *adjust_modulo_values()*, and assign them as coefficients of a list of polynomials with at most degree n, using *create_pol_ntru()*.

```
def text_to_base_p(text, base):
    base_p_encoding = ''
    for char in text:
        ascii_value = ord(char)  # to obtain the ASCII values
        base_p_digit = ''
        while ascii_value > 0: # to obtain the values in base p
            remainder = ascii_value % base
            base_p_digit = str(remainder) + base_p_digit
            ascii_value //= base
        base_p_encoding += base_p_digit.zfill(math.ceil(math.log(255,
           base)))  # Pad with zeros if necessary
    return base_p_encoding

def adjust_modulo_values(values, modulo):
    adjusted_values = []
    for value in values: # to adjust the values to be between -p/2+1 and
        p/2
        if value > modulo // 2:
            value -= modulo
        elif value < -modulo // 2 + 1:
            value += modulo
        adjusted_values.append(value)
    return adjusted_values

def create_pol_ntru(vect,n):
    R.<x> = PolynomialRing(ZZ, ['x'])
    f = sum(vect[0, i] * x^(n-i-1) for i in range(n)) # to create
       polynomials from vectors
    return f

def text_to_pol_ntru(text,p,n): # to code text to polynomials
    base_p_encoded = text_to_base_p(text, p)
    base_p_encoded_int = [int(val) for val in base_p_encoded]
    adjusted_values = adjust_modulo_values(base_p_encoded_int, p)
    num_vect = (len(adjusted_values) + n - 1) // (n)
    adjusted_values += [0] * (n * num_vect - len(adjusted_values))
    vectores_list = [np.array(adjusted_values[i * n:(i + 1) * n]).
       reshape(1, n) for i in range(num_vect)]
    pols = [create_pol_ntru(vect,n) for vect in vectores_list]
    return pols
```

Listing A.3: NTRU - Functions to convert text to polynomials

To generate a pair of public and private keys, this function generates two random polynomials F and G (A.6 and A.5, respectively), such that G has d_g ones and the same number of minus ones. The inverses of F modulo p and q, and also modulo $x^n - 1$ (denoted F_p and F_q) are then computed, resulting in the public key $H = F_q \cdot G \pmod q$ and the private keys F and F_p.

```
def find_inverse(p, n, f): # to find the inverse of f modulo p
    R.<x> = PolynomialRing(GF(p))
    ring_quotient = R.quotient(x^n - 1)
    f_quotient = ring_quotient(f) # f modulo x^n-1 and p
    inverse_quotient = f_quotient.inverse_of_unit()
    return ring_quotient(inverse_quotient)

def publicKey_ntru(n,p,q,d_f,d_g):
    f=poly_f(d_f,p,q,n) # private key from party 1
    f_p=find_inverse(p, n, f).mod(p) # private key from party 1
    f_q=find_inverse(q, n, f).mod(q)
    g=rand_poly_ntru(d_g,d_g,n)
    H=(f_q*g).mod(q) # public key from party 1
    return H, f, f_p
```

Listing A.4: NTRU - Functions to generate Public and Private Keys

This function creates polynomials with degree less than n, and coefficients: h number of ones, l number of minus ones, and $n - h - l$ number of zeros. h and l must be specified as arguments.

```
def rand_poly_ntru(h,l,n):
    R.<x> = PolynomialRing(ZZ, ['x'])
    num_ones, num_minus_ones = h, l
    coefficients = [0] * n
    for i in range(n):
        if num_ones > 0:
            coefficients[i] = 1
            num_ones -= 1
        elif num_minus_ones > 0:
            coefficients[i] = -1
            num_minus_ones -= 1
        else:
            break
    random.shuffle(coefficients)
    f = sum(coefficients[i] * x^i  for i in range(n))
    return f
```

Listing A.5: NTRU - Function to generate random polynomials

This function ensures the created polynomial f has d_f ones and $d_f - 1$ minus ones as coefficients and guarantees the polynomial has an inverse modulo p and q. If not, it regenerates the polynomial.

```
def invertible(p, n, f):
    R.<x> = PolynomialRing(GF(p))
    ring_quotient = R.quotient(x^n - 1)
    f_quotient = ring_quotient(f)
    return f_quotient.is_unit()

def poly_f(d_f,p,q,n):
    f=rand_poly_ntru(d_f,d_f-1,n) # creates f with d_f ones and d_f-1
       minus one
    if (invertible(p, n, f) and invertible(q, n, f)):
        return f
    else:
        return poly_f(d_f,p,q,n)
```

Listing A.6: NTRU - Functions to generate invertible polynomials

To decrypt, the private keys are used to calculate $B = (F \cdot E) \pmod q$, then the coefficients are adjusted to be between $-q/2 + 1$ and $q/2$ with *pol_adjust_module_ntru()*. Finally, $m = F_p \cdot B \pmod p$ gives the decrypted polynomials, which can be decoded back to the message.

```
def pol_adjust_module_ntru(poly, modulo,n):
    R.<x> = PolynomialRing(ZZ, ['x'])
    if isinstance(poly, int):
        coeffs = [poly]
    else:
        poly = Poly(str(poly))
        coeffs = poly.all_coeffs()
    adjusted_coeffs = adjust_modulo_values(coeffs, modulo)
    adjusted_expr = sum(int(c) * x^i for i, c in enumerate(
       adjusted_coeffs[::-1]))
    return adjusted_expr

def decrypt_ntru(f,f_p,E,p,q,n):
        B = (f * E).mod(q)
        B = pol_adjust_module_ntru(B,q)
        A = B.mod(p)
        m = (f_p*A).mod(p) # party 1 decrypts with its private keys
        return m
```

Listing A.7: NTRU - Decryption function

With the *pol_to_text_ntru()* function, we reverse A.3 and retrieve the original message.

```
def modulo_p_list(input_list,p):
    if p == 0:
        raise ValueError("p cannot be null.")
    return [x % p if x != 0 else 0 for x in input_list]

def base_p_list_to_text(base_p_encoding, base):
    text = ''
    num_digits = math.ceil(math.log(255, base))
    i = 0
    while i < len(base_p_encoding):
        base_p_digit = ''.join(str(x) for x in base_p_encoding[i:i+
            num_digits])
        ascii_value = int(base_p_digit, base)
        char = chr(ascii_value)
        text += char
        i += num_digits
    return text

def pol_to_text_ntru(pol_list,p,n):
    coeffs_all_vect = []
    for pol in pol_list:
        if (pol in [-1,0,1]):
            component_coeffs=[component]
        else:
            poly = Poly(str(pol))
            component_coeffs = poly.all_coeffs()
        num_zeros = n - len(component_coeffs)
        if num_zeros > 0:
            component_coeffs = [0] * num_zeros + component_coeffs
        coeffs_all_vect.extend(component_coeffs)
    adjusted_values=modulo_p_list(coeffs_all_vect,p)
    reconstructed_text = base_p_list_to_text(adjusted_values, p)
    return reconstructed_text
```

Listing A.8: NTRU - Function to convert polynomials back to text

A.2 QTRU Code

For QTRU, since we use quaternions in this system, we began by defining them in a class, as shown in A.9. We also defined arithmetic operations such as addition, multiplication, etc., along with the conjugate, norm, and modulo operations to make the coefficients of a quaternion modulo some number. Additionally, we defined lift and poly_lift functions to lift the polynomials from the modulo they were in.

SageMath does not allow direct operations between elements unless they belong to the same modulus, and it does not provide clear error messages in such cases. This limitation has posed challenges in writing the program, resulting in multiple layers of functions to work around it.

```
from sympy import symbols, expand
class Quaternion:
    def __init__(self, a, b, c, d):
        self.a = a
        self.b = b
        self.c = c
        self.d = d

    def __repr__(self):
        return f"({self.a}) + ({self.b})i + ({self.c})j + ({self.d})k"

    def __add__(self, other):
        return Quaternion(self.a + other.a, self.b + other.b, self.c +
            other.c, self.d + other.d)

    def __sub__(self, other):
        return Quaternion(self.a - other.a, self.b - other.b, self.c -
            other.c, self.d - other.d)

    def __mul__(self, other):
        a = self.a * other.a - self.b * other.b - self.c * other.c -
            self.d * other.d
        b = self.a * other.b + self.b * other.a + self.c * other.d -
            self.d * other.c
        c = self.a * other.c - self.b * other.d + self.c * other.a +
            self.d * other.b
        d = self.a * other.d + self.b * other.c - self.c * other.b +
            self.d * other.a
        return Quaternion(a, b, c, d)

    def mul_esc(self, num):
        return Quaternion(self.a*num, self.b*num, self.c*num, self.d*num
            )
```

```
27
28      def conjugate(self):
29          return Quaternion(self.a, -self.b, -self.c, -self.d)
30
31      def norm(self):
32          a, b, c, d = self.a, self.b, self.c, self.d
33          return a^2 + b^2 + c^2 + d^2
34
35      def mod(self,p,n):
36          if p==0:
37              R.<x> = PolynomialRing(ZZ)
38          else:
39              R.<x> = PolynomialRing(GF(p))
40          R_p_line = R.quotient(x^n - 1)
41          return Quaternion(R_p_line(self.a), R_p_line(self.b), R_p_line(
                self.c),R_p_line(self.d))
42
43      def expand(self):
44          return Quaternion(expand(self.a), expand(self.b), expand(self.c)
                , expand(self.d))
45
46      def lift(self):
47          return Quaternion((self.a).lift(), (self.b).lift(),(self.c).lift
                (),(self.d).lift())
48
49      def poly_lift(self):
50          R.<x> = PolynomialRing(ZZ, ['x'])
51          return Quaternion(R(Poly(self.a.lift())), R(Poly(self.b.lift()))
                ,R(Poly(self.c.lift())),R(Poly(self.d.lift())))
```

Listing A.9: QTRU - Quaternions class

Similar to NTRU, in QTRU we start by defining n, p, q, and also d_f, d_g and d_r (A.10), as parameters to perform A.11. We also choose our text to be encrypted. The message is then encoded letter by letter by taking its ASCII code modulo p and using these values as coefficients for the polynomials in each component of the quaternions. This process results in a list of polynomial quaternions, where each polynomial has at most degree n (A.12).

```
1 qtru_system(7,127,3,2,2,2,"QTRU")
```

Listing A.10: QTRU - Initialization and Encryption proccess

Once we have the list of quaternions, we need to generate a corresponding number of public keys from the first party. The first party generates each of these public keys and the corresponding private key (A.13). This involves generating two random quaternions, F and G (A.15 and A.14, respectively), and computing the inverse of F modulo p and q.

For the second party to perform the encryption, for each quaternion from the message to be encrypted, it generates a random quaternion R with d_r number of ones and minus ones (A.14). The list of encrypted quaternions is then given by $E = p \cdot (H \cdot R) + M \pmod q$.

Finally, the first party decrypts the message (A.16), converts the quaternions back to text (A.17), and retrieves the original message.

```
def qtru_system(n,q,p,d_f,d_g,d_r,text):
    quaternions = text_to_quaternion_qtru(text,p,n) #calculated by party
        2
    keys=len(quaternions) # party 2 gives to party 1 to calculate all
       key pairs
    publicKeys=[]
    privateKeys=[]
    for i in range(keys):
        H, f, f_p=publicKey_qtru(n,p,q,d_f,d_g)
        publicKeys.append(H) # party 1 gives to party 2
        privateKeys.append([f,f_p]) # kept a secret by party 1
    encrypt=[]
    for i in range(keys):
            M=quaternions[i].mod(0,n)
            r=rand_poly_qtru(d_r,d_r,n).mod(0,n)
            H=publicKeys[i]
            aux=(H*r).mul_esc(p)
            E=(aux+M).mod(q,n) # party 2 calculates the encryption and
               gives to party 1
            encrypt.append(E)
    m=[]
    for i in range(keys):
        M=decrypt_qtru(privateKeys[i][0],privateKeys[i][1],encrypt[i],p,
           q,n)
        m.append(M) # party 1 decrypts the message with its private keys
    reconstructed_text = quaternion_to_text_qtru(m,p,n) # party 1
       decodes the message and recover the message
```

Listing A.11: QTRU - Initialization and Encryption proccess

Firstly, every time we create a quaternion with polynomials in a function we need to define $R.<x>$ so that Sagemath defines x as a generator and the coefficients of the polynomials as integers.

Using *text_to_quaternion_qtru()*, we convert our message to base p with *text_to_base_p()*, adjust the values to be between $-p/2+1$ and $p/2$, and assign them to a list of quaternions with polynomials as coefficients of the quaternions, where the coefficients of each polynomial represent the message encoded in base p (adjusted).

```
def text_to_base_p(text, base):
    base_p_encoding = ''
    for char in text:
        ascii_value = ord(char) # to obtain the ASCII values
        base_p_digit = ''
        while ascii_value > 0: # to obtain the values in base p
            remainder = ascii_value % base
            base_p_digit = str(remainder) + base_p_digit
            ascii_value //= base
        base_p_encoding += base_p_digit.zfill(math.ceil(math.log(255,
           base))) # pad with zeros if necessary
    return base_p_encoding

def adjust_modulo_values(values, modulo):
    adjusted_values = []
    for value in values: # to adjust the values to be between -p/2+1 and
        p/2
        if value > modulo // 2:
            value -= modulo
        elif value < -modulo // 2 + 1:
            value += modulo
        adjusted_values.append(value)
    return adjusted_values

def create_quaternion_qtru(matrix,n):
    R.<x> = PolynomialRing(ZZ, ['x'])
    f = []
    for row in range(4):
        f_row = sum(matrix[row, i] * x**(n-i-1) for i in range(n)) # to
           create quaternions of polynomials from a matrix
        f.append(f_row)
    return Quaternion(f[0], f[1], f[2], f[3])

def text_to_quaternion_qtru(text,p,n): # to code text to quaternions
    base_p_encoded = text_to_base_p(text, p)
    base_p_encoded_int = [int(val) for val in base_p_encoded]
    adjusted_values = adjust_modulo_values(base_p_encoded_int, p)
    num_matrices = (len(adjusted_values) + 4*n - 1) // (4*n)
    adjusted_values+=[0] * (4 * n * num_matrices - len(adjusted_values))
    matrices_list = [np.array(adjusted_values[i * 4 * n:(i + 1) * 4 * n
       ]).reshape(4, n) for i in range(num_matrices)]
    quaternions = [create_quaternion_qtru(matrix,n) for matrix in
       matrices_list]
    return quaternions
```

Listing A.12: QTRU - Functions to convert text to quaternions

To generate a pair of public and private keys, this function generates two random quaternions F and G (A.15 and A.14, respectively), such that each polynomial of the quaternion components of G has d_g ones and the same number of minus ones. The inverses of F modulo p and q (denoted F_p and F_q) are then computed, resulting in the public key $H = F_q \cdot G \pmod q$ and the private keys F and F_p.

```
def find_inverse(p, n, f): # to find the inverse of f modulo p
    R.<x> = PolynomialRing(GF(p))
    ring_quotient = R.quotient(x^n - 1)
    f_quotient = ring_quotient(f)
    inverse_quotient = f_quotient.inverse_of_unit()
    return ring_quotient(inverse_quotient)

def publicKey_qtru(n,p,q,d_f,d_g):
    R.<x> = PolynomialRing(ZZ, ['x'])
    f=poly_f(d_f,p,q,n) # private key from party 1
    f_p=(f.conjugate().mod(p,n).mul_esc(find_inverse(p, n, f.norm()))).
        mod(p,n) # private key from party 1
    f_q=((f.conjugate()).mod(q,n).mul_esc(find_inverse(q, n, f.norm())))
        .mod(q,n)
    g=rand_poly_qtru(d_g,d_g,n)
    H=(f_q.lift()*g).mod(0,n) # public key from party 1
    return H, f, f_p
```

Listing A.13: QTRU - Functions to generate Public and Private Keys

This function *rand_poly_qtru()* creates quaternions with each component being a polynomial with degree less than n, and coefficients: h number of ones, l number of minus ones, and $n - h - l$ number of zeros. h and l must be specified as arguments

```
def rand_poly_qtru(h,l,n):
    R.<x> = PolynomialRing(ZZ, ['x'])
    num_ones = h
    num_minus_ones = l
    coefficients = [0] * n
    for i in range(n):
        if num_ones > 0:
            coefficients[i] = 1
            num_ones -= 1
        elif num_minus_ones > 0:
            coefficients[i] = -1
            num_minus_ones -= 1
        else:
            break
    f = []
    for _ in range(4):
        random.shuffle(coefficients)
        f.append(sum(coefficients[i] * x**i for i in range(n)))
    f0, f1, f2, f3 = f
    Q = Quaternion(f0, f1, f2, f3) # quaternion with polynomials as
       coefficients
    return Q
```

Listing A.14: QTRU - Function to generate random quaternions

This function *poly_f()* ensures the created quaternion has d_f ones and $d_f - 1$ minus ones as coefficients of the polynomials of each component and guarantees the quaternion has an inverse modulo p and q. If not, it regenerates the quaternion.

```
def invertible(p, n, f):
    R.<x> = PolynomialRing(GF(p))
    ring_quotient = R.quotient(x^n - 1)
    f_quotient = ring_quotient(f)
    return f_quotient.is_unit()

def poly_f(d_f,p,q,n):
    f=rand_poly_qtru(d_f,d_f-1,n) # creates f with d_f ones and d_f-1
minus one
    if (invertible(p, n, f.norm()) and invertible(q, n, f.norm())):
        return f
    else:
        return poly_f(d_f,p,q,n)
```

Listing A.15: QTRU - Functions to generate invertible quaternions

To decrypt, the private keys are used to calculate the quaternion $B = (F \cdot E) \pmod q$ with *decrypt_qtru()*, then the coefficients of the polynomial components are adjusted to be between $-q/2 + 1$ and $q/2$, with *quat_adjust_module_qtru()*. Finally, $m = F_p \cdot B \pmod p$ gives the decrypted quaternions, which can be decoded back to the message in A.17.

```
def quat_adjust_module_qtru(quat, modulo,n):
    R.<x> = PolynomialRing(ZZ, ['x'])
    pols = []
    for poly in [quat.a, quat.b, quat.c, quat.d]:
        if isinstance(poly, int):
            coeffs = [poly]
        else:
            poly = Poly(str(poly))
            coeffs = poly.all_coeffs()
        adjusted_coeffs = adjust_modulo_values(coeffs, modulo)
        adjusted_expr = sum(int(c) * x^i for i, c in enumerate(
           adjusted_coeffs[::-1]))
        pols.append(adjusted_expr)
    return Quaternion(((pols[0])), ((pols[1])), ((pols[2])), ((pols[3]))
       )

def decrypt_qtru(f,f_p,E,p,q,n):
        B = (f * E).mod(q,n)
        B = quat_adjust_module_qtru(B.lift(),q,n)
        A = B.mod(p,n)
        m = (f_p*A).mod(p,n).lift() # party 1 decrypts with its private
           keys
        return m
```

Listing A.16: QTRU - Decryption function

With the *quaternion_to_text_qtru()* function, we reverse the process in A.12 and retrieve the original message.

```
def modulo_p_list(input_list,p):
    if p == 0:
        raise ValueError("p cannot be null.")
    return [x % p if x != 0 else 0 for x in input_list]

def base_p_list_to_text(base_p_encoding, base):
    text = ''
    num_digits = math.ceil(math.log(255, base))
    i = 0
    while i < len(base_p_encoding):
        base_p_digit = ''.join(str(f) for f in base_p_encoding[i:i+
            num_digits])
        ascii_value = int(base_p_digit, base)
        char = chr(ascii_value)
        text += char
        i += num_digits
    return text

def quaternion_to_text_qtru(quaternions_list,p,n):
    coeffs_all_quaternions = []
    for quaternion in quaternions_list:
        coeffs = []
        for component in [quaternion.a,quaternion.b,quaternion.c,
            quaternion.d]:
            if (component in [-1,0,1]):
                component_coeffs=[component]
            else:
                poly = Poly(str(component))
                component_coeffs = poly.all_coeffs()
            num_zeros = n - len(component_coeffs)
            if num_zeros > 0:
                component_coeffs = [0] * num_zeros + component_coeffs
            coeffs.extend(component_coeffs)
        coeffs_all_quaternions.extend(coeffs)
    adjusted_values=modulo_p_list(coeffs_all_quaternions,p)
    reconstructed_text = base_p_list_to_text(adjusted_values, 3)
    return reconstructed_text
```

Listing A.17: QTRU - Function to convert quaternions back to text

Bibliography

[1] Aminullah Abidin and Jan-Åke Larsson. Direct proof of security of wegman-carter authentication with partially known key. *Quantum Information Processing*, 13(10), 2014. doi: 10.1007/s11128-013-0641-6. URL https://doi.org/10.1007/s11128-013-0641-6.

[2] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How diffie-hellman fails in practice. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015. URL https://api.semanticscholar.org/CorpusID:347988.

[3] Khadijeh Bagheri, Mohammad-Reza Sadeghi, and Daniel Panario. A non-commutative cryptosystem based on quaternion algebras. *Designs, Codes and Cryptography*, 2018. ISSN 1573-7586. doi: 10.1007/s10623-017-0451-4. URL https://doi.org/10.1007/s10623-017-0451-4.

[4] Daniel J. Bernstein. *Introduction to post-quantum cryptography*. Springer Berlin Heidelberg, 2009. ISBN 978-3-540-88702-7. doi: 10.1007/978-3-540-88702-7_1. URL https://doi.org/10.1007/978-3-540-88702-7_1.

[5] Daniel J. Bernstein and Tanja Lange. Post-quantum cryptography. *Nature*, 2017. ISSN 1476-4687. doi: 10.1038/nature23461. URL https://doi.org/10.1038/nature23461.

[6] Narasimham Challa and Jayaram Pradhan. Performance analysis of public key cryptographic systems rsa and ntru. 2007. URL https://api.semanticscholar.org/CorpusID:7558890.

[7] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on post-quantum cryptography. Internal Report 8105, National Institute of Standards and Technology, April 2016. URL http://dx.doi.org/10.6028/NIST.IR.8105.

[8] Mariana Durcheva and Kristian Karailiev. New public key cryptosystem based on quaternions. volume 1910, 12 2017. doi: 10.1063/1.5014008.

[9] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability*. The Chance Project, Hanover, N.H., 2006. URL https://worldcat.org/title/70601649.

[10] Jeffrey Hoffstein and Joseph H. Silverman. *An Introduction to Mathematical Cryptography*. Undergraduate Texts in Mathematics. Springer Science+Business Media, LLC, 2008. ISBN 978-0-387-77993-5. doi: 10.1007/978-0-387-77994-2. Library of Congress Control Number: 2008923038.

[11] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. Ntru: A ring-based public key cryptosystem. In Joe P. Buhler, editor, *Algorithmic Number Theory*. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-69113-6.

[12] Thomas W. Hungerford. *Rings*. Springer New York, 1974. ISBN 978-1-4612-6101-8. doi: 10.1007/978-1-4612-6101-8_4. URL https://doi.org/10.1007/978-1-4612-6101-8_4.

[13] Jorge Kamlofsky, Juan Pedro Hecht, O. Izzi, and S. Masuh. *A Diffie-Hellman Compact Model Over Non-Commutative Rings Using Quaternions*. 11 2015. doi: 10.13140/RG.2.1.4063.1760.

[14] K. Sundara Krishnan. Symmetric key cryptosystem. 2004. URL https://api.semanticscholar.org/CorpusID:18666242.

[15] Greg Kuperberg. https://complexityzoo.net/Complexity_Zoo. Accessed on June 27, 2024.

[16] T. Y. Lam. *Introduction to Division Rings*. Springer US, 1991. ISBN 978-1-4684-0406-7. doi: 10.1007/978-1-4684-0406-7_5. URL https://doi.org/10.1007/978-1-4684-0406-7_5.

[17] T. Y. Lam. *Wedderburn-Artin Theory*. Springer New York, 2001. ISBN 978-1-4419-8616-0. doi: 10.1007/978-1-4419-8616-0_1. URL https://doi.org/10.1007/978-1-4419-8616-0_1.

[18] E. Malekian, A. Zakerolhosseini, and A. Mashatan. *QTRU: quaternionic version of the NTRU public-key cryptosystems*. Iranian Society of Cryptology, 2011. doi: 10.22042/isecure.2015.3.1.3. URL https://www.isecure-journal.com/article_39184.html.

[19] Jintai Ding Matthew Campagna, Lidong Chen. *Quantum Safe Cryptography and Security; An Introduction, Benefits, Enablers and Challenges*. ETSI, 2020. ISBN 979-10-92620-03-0.

[20] Marius Iulian Mihailescu and Stefania Loredana Nita. Asymmetric encryption schemes. *Cryptography and Cryptanalysis in MATLAB*, 2021. URL https://api.semanticscholar.org/CorpusID:239254289.

[21] Tristan Moore. *Quantum Computing and Shor's Algorithm*. 2016. URL https://api.semanticscholar.org/CorpusID:204958873.

[22] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 1999. doi: 10.1137/S0036144598347011. URL https://doi.org/10.1137/S0036144598347011.

[23] John Voight. *Quaternion Algebras*. Graduate Texts in Mathematics. Springer, 1 edition, 2021. ISBN 978-3-030-56692-0. doi: 10.1007/978-3-030-56694-4. URL https://doi.org/10.1007/978-3-030-56694-4. Published: 29 June 2021 (hardcover), 30 June 2022 (softcover), 28 June 2021 (eBook).