POLITECNICO
MILANO 1863

Alekhnovich's Cryptosystem

PII

Course Instructor: Ing. Alessandro Barenghi

August 18, 2024

Introduction

In the modern era, data has become an invaluable resource, akin to oil in its capacity to drive economies and influence global dynamics. With the exponential growth of digital data, the need for robust data protection mechanisms has become paramount. This necessity is underscored by the emergence of quantum computers, which pose a significant challenge to traditional cryptographic methods.

Quantum computers, leveraging principles of quantum mechanics such as superposition and entanglement, possess the potential to perform computations at speeds far surpassing classical computers. This capability directly threatens current cryptographic systems, particularly those based on the hardness of problems like integer factorization and discrete logarithms, which are vulnerable to quantum algorithms such as Shor's algorithm.

However, the primary challenge in addressing quantum threats lies not in the immediate danger but in the slow and complex transition from quantum-vulnerable cryptographic schemes to quantum-resistant ones. The adoption of post-quantum cryptographic systems must begin well before quantum computers become practically viable, due to the time-consuming nature of cryptographic transitions and the widespread reliance on current systems.

This report delves into Alekhnovich's cryptosystem, a candidate for post-quantum cryptography, which bases its security on the hardness of decoding random linear codes—an NP-hard problem believed to be resistant to quantum attacks. We will explore the theoretical underpinnings of this cryptosystem, discuss its implementation details, and evaluate its potential to serve as a robust defense in the quantum era.

Contents

Chapter 1

Theoretical Fundamentals

To address the challenges posed by quantum computers, the computer science community has long been engaged in the quest for alternatives to current symmetric and asymmetric encryption algorithms. Concerning the latter, various proposals have been presented to the public and the community, yet only a few have been recognized as valid and genuinely effective against quantum computers. As for other algorithms, we have thus far been unable to demonstrate either their efficacy or inefficacy in the face of quantum computers. Among these is the Alekhnovich cryptosystem, the focus of our exploration in this treatise.

To delve deeper into this cryptosystem, it is essential to introduce several concepts from the fields of coding theory, geometry, and linear algebra. This introduction will enhance the comprehensibility of its implementation, both in terms of logic and procedures.

1.1 Linear Algebra

Linear algebra, a cornerstone of mathematics, is the study of vector spaces and their transformations. It is fundamental in a wide range of applications, including the solution of systems of linear equations, transformations of multidimensional objects in space, and much more. To undertake these analyses, linear algebra employs various mathematical constructs, including vector spaces, linear equations, and matrices.

1.1.1 Matrices

Matrices, rectangular arrays of numbers, symbols, or expressions arranged in rows and columns, play a crucial role in linear algebra. They provide a structured way to represent vectors, linear transformations, and systems of linear equations. Matrices allow us to handle complex operations such as addition, multiplication, and transposition, thereby enabling the solution of diverse mathematical problems.

Matrix. A **matrix** is a rectangular array of numbers, symbols, or expressions arranged in rows and columns. It is denoted as $A = [a_{ij}]$, where a_{ij} represents the element in the i-th row and j-th column of matrix A.

Matrix Addition. The sum of two matrices A and B of the same dimensions is defined as a matrix C, where each entry c_{ij} in C is the sum of the corresponding entries a_{ij} and b_{ij} in matrices A and B. Mathematically,

$$C = [c_{ij}] \text{ where } c_{ij} = a_{ij} + b_{ij}.$$

Matrix Multiplication. The product of two matrices A and B is a matrix C, where each entry c_{ij} is the sum of the products of the corresponding entries in the i-th row of A and the j-th column of B. This operation is defined only when the number of columns in A matches the number of rows in B. Mathematically,

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}.$$

For example, if:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix},$$

Then,

$$C = A \times B = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 41 & 50 \end{bmatrix}.$$

1.1.2 Fields and Vector Spaces

In linear algebra, fields and vector spaces form the foundational building blocks that underpin much of the theory and applications. A field provides the necessary structure for performing arithmetic operations, while a vector space is a collection of vectors that can be added together and multiplied by scalars from a field. Understanding these concepts is crucial for delving deeper into more advanced topics in linear algebra.

Fields are algebraic structures that allow for the operations of addition, subtraction, multiplication, and division (except by zero). They provide the scalar elements that are used in the definition of vector spaces.

Vector Spaces, on the other hand, are collections of vectors that can be scaled and added together in a manner consistent with the rules of the underlying field. These spaces are essential for describing linear transformations and solving systems of linear equations.

Field. A **field** is a set $\mathbb{F}$ equipped with two operations: addition ($+$) and multiplication ($\cdot$), satisfying the following properties:

1. **Closure:** For all $a, b \in \mathbb{F}$:
$$a + b \in \mathbb{F} \quad \text{and} \quad a \cdot b \in \mathbb{F}.$$

2. **Associativity:** For all $a, b, c \in \mathbb{F}$:
$$(a + b) + c = a + (b + c) \quad \text{and} \quad (a \cdot b) \cdot c = a \cdot (b \cdot c).$$

3. **Commutativity:** For all $a, b \in \mathbb{F}$:
$$a + b = b + a \quad \text{and} \quad a \cdot b = b \cdot a.$$

4. **Distributivity:** For all $a, b, c \in \mathbb{F}$:
$$a \cdot (b + c) = (a \cdot b) + (a \cdot c).$$

5. **Identity Elements:** There exist elements $0 \in \mathbb{F}$ and $1 \in \mathbb{F}$ such that for all $a \in \mathbb{F}$:
$$a + 0 = a \quad \text{and} \quad a \cdot 1 = a.$$

6. **Inverses:** For every $a \in \mathbb{F}$, there exists an element $-a \in \mathbb{F}$ such that:

$$a + (-a) = 0.$$

For every non-zero $a \in \mathbb{F}$, there exists an element $a^{-1} \in \mathbb{F}$ such that:

$$a \cdot a^{-1} = 1.$$

An example of a field is $\mathbb{Z}_2$, the set of integers $\bmod 2$, with elements $\{0, 1\}$, where the operations of addition and multiplication are defined as follows:

$+$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

Vector Space. A **vector space** V over a field $\mathbb{F}$ is a set of vectors equipped with two operations: vector addition and scalar multiplication, satisfying the following properties:

1. **Closure:** For all $u, v \in V$ and $\alpha \in \mathbb{F}$:

$$u + v \in V \quad \text{and} \quad \alpha \cdot u \in V.$$

2. **Associativity of Addition:** For all $u, v, w \in V$:

$$(u + v) + w = u + (v + w).$$

3. **Commutativity of Addition:** For all $u, v \in V$:

$$u + v = v + u.$$

4. **Additive Identity:** There exists a zero vector $0 \in V$ such that for all $u \in V$:

$$u + 0 = u.$$

5. **Additive Inverses:** For every $u \in V$, there exists an element $-u \in V$ such that:

$$u + (-u) = 0.$$

6. **Distributivity of Scalar Multiplication:** For all $\alpha, \beta \in \mathbb{F}$ and $u \in V$:

$$\alpha \cdot (u + v) = (\alpha \cdot u) + (\alpha \cdot v).$$

7. **Distributivity of Scalar Addition:** For all $\alpha, \beta \in \mathbb{F}$ and $u \in V$:

$$(\alpha + \beta) \cdot u = (\alpha \cdot u) + (\beta \cdot u).$$

8. **Multiplicative Identity:** For every $u \in V$:

$$1 \cdot u = u.$$

An example of a vector space is $\mathbb{Z}_2^n$, where n is the dimension of the space, and each vector consists of n elements from $\mathbb{Z}_2$. The operations of vector addition and scalar multiplication are performed element-wise.

1.2 Coding Theory

Coding theory refers to the analysis of code properties and its numerous applications. In the realm of communications and information processing, the concept of a code pertains to a set of rules designed to transform data, such as text, numbers, or images, into other forms that are more suitable for storage, transmission, or encryption.

Code. A code is a set of vectors that represent the encoded form of data. It is not merely a function but rather an ensemble of image vectors over a chosen alphabet.

Hamming Distance. The Hamming distance is a measure used to quantify the dissimilarity between two strings of equal length. It calculates the minimum number of substitutions required to change one string into another by altering individual elements.

For instance, consider two strings: 101010 and 111011. Their Hamming distance is 2 because, to convert the first string into the second, two substitutions are needed: changing the second and fifth elements from 0 to 1.

1.2.1 Channel Coding

The detection and correction of errors in a code ensure the integrity of data information, whether transmitted over noise-exposed channels or for simple storage purposes. The principle is straightforward: redundancy, the addition of seemingly unnecessary information, is essential for anyone wishing to verify data accuracy or recover lost or corrupted information fragments. Some codes, such as the repetition code, Hamming code, and extended Hamming code, allow for decoding algorithms to correct errors.

- **Repetition Code:** A coding scheme that repeats blocks of bits n times, determining the correct value between 0 and 1 depending on which repeats more than $n/2$ times.
- **Hamming Code:** A code with a minimum Hamming distance that can detect up to $d - 1$ errors and correct up to $\lfloor (d-1)/2 \rfloor$ errors.
- **Extended Hamming Code:** A variant of the Hamming code that includes an additional parity bit, increasing the minimum distance and improving error detection capabilities.

1.2.2 Generator and Check Matrices

A linear code $\mathcal{C}$, which represents a subspace of $\mathbb{F}_q^n$, can be described by a generator matrix G, where the rows of G form a basis for $\mathcal{C}$. This matrix allows for the transformation of information words into codewords through matrix multiplication.

To ensure that codewords belong to $\mathcal{C}$, a check matrix H, also known as a parity check matrix, is constructed. The kernel of H defines the code $\mathcal{C}$, meaning any valid codeword $\mathbf{c}$ satisfies $H\mathbf{c}^T = \mathbf{0}$.

When G is in standard form, $G = [\mathbb{I}_k \mid P]$, where $\mathbb{I}_k$ is the $k \times k$ identity matrix and P is a $k \times (n-k)$ matrix, the corresponding check matrix H is given by $H = [-P^T \mid \mathbb{I}_{n-k}]$. This ensures that the dot product of any codeword with the rows of H yields the zero vector, validating the codeword as a member of $\mathcal{C}$.

Kernel. The **kernel** of a linear map (or the **null space**) is the set of all vectors in the domain that are mapped to the zero vector in the codomain. For a matrix A, the kernel is defined as:

$$\ker(A) = \{\mathbf{x} \in \mathbb{F}^n \mid A\mathbf{x} = \mathbf{0}\}.$$

Properties of the Kernel:

1. **Subspace:** The kernel is a subspace of the domain $\mathbb{F}^n$.
2. **Dimension:** The dimension of the kernel is called the **nullity** of A.
3. **Uniqueness of the Zero Vector:** The kernel always contains the zero vector.
4. **Relation to System of Linear Equations:** The kernel corresponds to the solution set of the homogeneous system $A\mathbf{x} = \mathbf{0}$.
5. **Invariant under Similarity Transformations:** The kernel is invariant under similarity transformations.

1.2.3 Cryptographic Coding

Cryptographic coding is a field dedicated to securing data integrity, confidentiality, and authentication. Encryption, a key component, converts readable data into secure, unreadable ciphertext, ensuring confidentiality. Decryption, its counterpart, allows authorized users with the correct key to access and transform ciphertext back into its original form for use.

Public-key cryptography, also known as asymmetric cryptography, stands out for its use of key pairs—a public key for encryption and a private key for decryption. This innovation simplifies secure communication by allowing individuals to openly share their public keys while keeping their private keys confidential. This concept plays a pivotal role in protecting digital systems in today's interconnected world.

Chapter 2

Alekhnovich's Cryptosystem

This paper does not delve into the intricate details of the Alekhnovich algorithm or its potential advantages against quantum computers compared to existing algorithms. However, to provide context for further discussion, a brief overview of its operation is included.

2.1 The Algorithm

The Alekhnovich cryptosystem[1] is based on hard problems in coding theory, particularly the problem of decoding random linear codes, which is believed to be hard for both classical and quantum computers. The algorithm begins by generating two random matrices: $\mathbf{A} \in \mathbb{Z}_2^{k\times n}$ and $\mathbf{S} \in \mathbb{Z}_2^{l\times k}$. Here, $\mathbf{A}$ acts as the generator matrix of a random linear code, and S serves as a scrambling matrix that transforms the code generated by $\mathbf{A}$.

Subsequently, a third random matrix $\mathbf{E} \in \mathbb{Z}_2^{l\times n}$ is generated, where each row of $\mathbf{E}$ is a random vector with a Hamming weight of $t = \sqrt{n}$. The matrix $\mathbf{Y}$ is then defined as $\mathbf{Y} = \mathbf{SA} + \mathbf{E}$. In this context, $\mathbf{Y}$ can be interpreted as a noisy version of the matrix $\mathbf{SA}$, where the noise is introduced by $\mathbf{E}$. This step ensures that the problem of distinguishing between random vectors and codewords of the code defined by $\mathbf{A}$ becomes computationally hard.

The public key of the algorithm consists of the matrices $\mathbf{Y} \in \mathbb{Z}_2^{l\times n}$ and $\mathbf{A} \in \mathbb{Z}_2^{k\times n}$, which are shared with anyone wishing to send encrypted messages. The plaintext space is $\mathbf{m} \in \mathcal{C} \subset \{0,1\}^l$, where $\mathcal{C}$ is an error-correcting code characterized by its length, dimension, and minimum distance. The code $\mathcal{C}$ is crucial as it allows for the recovery of the message even if it has been corrupted by noise.

The encryption and decryption process is as follows:

- **Encryption:**
  1. The sender generates a random t-weight vector $\mathbf{e} \in \mathbb{F}_2^n$.
  2. The sender computes the vectors $\mathbf{Ae}^T$ and $\mathbf{Ye}^T$, where $\mathbf{A}$ and $\mathbf{Y}$ are the public key matrices.
  3. The encrypted message is then formed by adding the plain text message $\mathbf{m}$ to $\mathbf{Ye}^T$:

$$\mathtt{C}(\mathbf{m}) = (\mathbf{Ae}^T, \mathbf{m} + \mathbf{Ye}^T).$$

- **Decryption:**

1. The receiver multiplies the first part of the encrypted message, $\mathbf{A}\mathbf{e}^T$, by the secret key $\mathbf{S}$ to obtain $\mathbf{S}\mathbf{A}\mathbf{e}^T = \mathbf{Y}\mathbf{e}^T - \mathbf{E}\mathbf{e}^T$.
2. The receiver then subtracts this from the second part of the encrypted message, $\mathbf{m}+\mathbf{Y}\mathbf{e}^T$, yielding:
$$(\mathbf{m} + \mathbf{Y}\mathbf{e}^T) - (\mathbf{Y}\mathbf{e}^T - \mathbf{E}\mathbf{e}^T) = \mathbf{m} + \mathbf{E}\mathbf{e}^T.$$
3. Since $\mathbf{E}$ is sparse, the vector $\mathbf{E}\mathbf{e}^T$ is expected to have a low weight, and the error-correcting code $\mathcal{C}$ is employed to recover the original message $\mathbf{m}$ from $\mathbf{m} + \mathbf{E}\mathbf{e}^T$.

The security of the algorithm relies on the difficulty of decoding a random linear code, a problem that remains hard even when a certain number of errors (up to t) are introduced. The product of the matrix $\mathbf{E}$ and the vector e of weight t is likely to be zero due to the sparsity of $\mathbf{E}$ and $\mathbf{e}$. Specifically, the probability of overlap between non-zero entries is low, estimated as $P(\langle \mathbf{e}, \mathbf{E_i} \rangle = 0) \approx 1 - \frac{t^2}{n}$.

2.1.1 Parameters

The cryptosystem's implementation requires careful selection of parameters to ensure both security and efficiency. The matrix dimensions and message length must satisfy the condition

$$k + l < Rn,$$

where R is a constant less than 1. This inequality ensures that the system remains overdetermined, which is necessary for the security guarantees provided by the underlying hard problem.

An essential consideration in implementation is the choice of matrix dimensions that fit within the constraints of computer architecture. While selecting dimensions that are multiples of the architecture's word size (e.g., 64 bits) can improve computational efficiency, this can also introduce potential security risks by making the structure of the matrices more predictable. Therefore, it is recommended to select dimensions that are close to, but not exactly multiples of, 64 bits.

Additionally, the following key points should be noted:

- The weight of the error vector t is chosen as
$$t = \sqrt{l},$$
where l is the length of the message.
- The error probability of the transmission channel, based on the binary symmetric channel model, is
$$p = \frac{t^2}{n}.$$

For our implementation, the following parameter values will be used: $l = 1300$ *bits,* $t = 144$ *bits,* $k = 2^6$ *bits,* $n = 2^{17}$ *bits.* Moreover, to maintain the hardness of the decoding problem, it is crucial that the error weight t is high enough to ensure that the decoding problem remains computationally intractable.

2.2 Implementation

To facilitate secure message exchange, we must generate public and private keys. For efficient resource usage, we'll represent matrices with integers, treating individual bits as matrix cells. We will employ robust random number generation techniques using the following libraries:

- **librandombytes[2]:** generates cryptographically secure random seeds.
- **xoshiro256 PRNG[3]:** efficiently generates pseudo-random sequence of bits.

2.2.1 Helper Functions

For clarity and modularity, the following helper functions are introduced to handle bitwise operations, matrix transposition, and seed initialization, which are essential for the key generation and cryptographic processes.

Seed Initiation This function is crucial for preparing the seed used by the *xoshiro256* PRNG, ensuring that the sequences generated are cryptographically secure and unpredictable.

```
Algorithm 1: init_seed
  Result: Initialized seed vector s for pseudo random number generation
for $i$ in range(4):
    seed ← array
    randombytes(seed, 8)                          /* librandombytes API */
    for $j$ in range(8):
        $s_i \leftarrow s_i \ll 8$
        $s_i \leftarrow s_i \mid seed_j$           /* Bit-wise OR operation */
```

Bit Manipulation Functions To facilitate operations on individual bits within matrix cells, the following utility functions are defined:

- `fetch_bit(`e, k`)`: This function returns the bit at the k^{th} position of the element e.
- `shift_bit(`b, s`)`: This function shifts the bit b left by s positions.

Transposition To optimize bitwise operations on matrices, an efficient strategy for transposing individual bits is necessary. This involves converting rows of matrix into columns.

Algorithm 2: `matrix_transposed`

```
Input: $\mathbf{M} \in \mathbb{Z}_2^{l \times n}$
Output: $\mathbf{T} \in \mathbb{Z}_2^{m \times p}$
Data: $m = n \times \text{sizeof}$; $p = \frac{l}{\text{sizeof}}$
T ← matrix
for $i$ in range($m.rows$):
    for $j$ in range($m.columns$):
        for $k$ in range($sizeof$):
            $bit \leftarrow$ fetch_bit($m_{ij}, k$)
            $y \leftarrow j \times$ sizeof $+ k$
            $w \leftarrow \left\lfloor \frac{i}{\texttt{sizeof}} \right\rfloor$
            $s \leftarrow$ sizeof $- (i \bmod$ sizeof$) - 1$
            $t_{yw} \leftarrow t_{yw}$ | shift_bit(bit, $s$)
return T
```

Bit-wise AND & XOR Operations In this scenario, where individual matrix elements are represented by single bits rather than entire cells, it is essential to employ bitwise operations for the row-by-column product between matrices. Specifically, a bitwise AND operation is used to combine corresponding bits from the two matrices, followed by a bitwise XOR operation to accumulate the results. This approach ensures that the operations are correctly applied at the bit level.

Algorithm 3: `bax`

```
Input: $\mathbf{a}, \mathbf{v} \in \mathbb{Z}_2^l$
Output: result $\in \mathbb{Z}_2$
$result \leftarrow 0$
for $i$ in range($l$):
    $and \leftarrow a_i \,\&\, v_i$                              /* Bit-wise AND operation */
    $result \leftarrow result$ | $and$
$result \leftarrow$ count_ones(result) (mod 2)                  /* Simplified XOR */
return $result$;
```

2.2.2 Key Generation

The generation of keys is critical to the security of the cryptographic system. Here, we define a sequence of steps to generate both public and private keys efficiently while ensuring robustness against potential attacks. These keys are essential for encrypting and decrypting messages securely in subsequent processes.

Matrix A Following the referenced work, matrix A has dimensions $k \times n$ and contains random values. We'll use the following function to generate A after initializing the seed:

$$\mathbf{A} \leftarrow \texttt{random\_matrix}(a.rows,\ a.columns)$$

Algorithm 4: `random_matrix`

```
Input: rows, columns $\in \mathbb{N}$
Output: random matrix $\mathbf{M} \in \mathbb{Z}_2^{r \times c}$
1  init_seed()
2  $\mathbf{M} \leftarrow$ matrix
3  for $i$ in range($rows$):
4      for $j$ in range($columns$):
5          $m_{ij} \leftarrow$ xoshiro256.next()                /* Filling the matrix */
6  return $\mathbf{M}$
```

Note: If n represents the total number of bits, we can reduce **A**'s size as follows:

$$a.columns \leftarrow \frac{n}{s}$$

where s is the `size-of` the integer type chosen for our matrix.

Matrix S The same procedure is used for matrix **S**, simply changing the input variables:

$$\mathbf{S} \leftarrow \texttt{random\_matrix}(s.rows,\ s.columns)$$

Matrix E Matrix **E** has different requirements: a predetermined number of 1-bits (defined as weight t) must be randomly positioned within each row. The following function generates $e.rows$ arrays, each containing t randomly placed 1s:

$$\mathbf{E} \leftarrow \texttt{weighted\_matrix}(e.rows,\ e.columns,\ t)$$

Algorithm 5: `weighted_array`

```
Input: l, t $\in \mathbb{N}$
Output: random weighted vector $\mathbf{a} \in \mathbb{F}_2^l$
1  init_seed()
2  $\mathbf{a} \leftarrow$ array
3  for $count$ in range($t$):
4      repeat
5          $p \leftarrow$ xoshiro256.next()
6          $i \leftarrow\leftarrow \lfloor \frac{p}{\texttt{sizeof}} \rfloor$
7          $s \leftarrow p \pmod{sizeof}$
8      until fetch_bit($a_i, s$) $\neq 1$
9      $a_i = a_i$ | shift_bit($1, s$)          /* Set the bit at calculated position */
10 return $\mathbf{a}$
```

Matrix Y To compute matrix **Y**, we need to first transpose matrix **A** so that its rows become columns. Then, we perform a bitwise row-by-column multiplication between the transposed matrix $\mathbf{A}^T$ and matrix **S**. Finally, we add matrix **E** to the resulting matrix to obtain the final matrix **Y**. The following pseudocode describes the process:

Algorithm 6: `compute_pub`

```
Input: $\mathbf{A} \in \mathbb{Z}_2^{k\times n}, \mathbf{S} \in \mathbb{Z}_2^{l\times k}, \mathbf{E} \in \mathbb{Z}_2^{l\times n}$
Output: $\mathbf{Y} \in \mathbb{Z}_2^{l\times n}$
$\mathbf{T} \leftarrow$ matrix_transposed($\mathbf{A}$)
for $i$ in range($l$):
    for $j$ in range($n$):
        $y_{ij} \leftarrow$ bax($\mathbf{S}_j, \mathbf{T}_i$)          /* Bit-wise row-by-column product using BAX */
for $i$ in range($l$):
    for $j$ in range($n$):
        $y_{ij} \leftarrow y_{ij} \mid e_{ij}$
return $\mathbf{Y}$
```

2.2.3 Encryption

Following the generation of cryptographic keys, these keys are employed in the encryption and decryption processes. For the encryption of messages, we select an appropriate message element from the cryptographic system, denoted as $\mathbf{m} \in \mathcal{C} \subseteq \{0,1\}^l$. Additionally, it is essential to generate a random vector of fixed weight t, which can be achieved using the function defined previously:

$$\mathbf{e} \leftarrow \texttt{weighted\_array}(n, t)$$

The encryption process itself is straightforward, requiring two function calls:

1. **nnc** $\leftarrow$ `compute_nonce`(A, e)[1], which computes the row-by-column product, treating $\mathbf{e}$ as a single-column matrix.

2. **cmp** $\leftarrow$ `cipher`$(\mathbf{Y}, \mathbf{e}, \mathbf{m})$, which not only performs a row-by-column product similar to the previous step but also integrates the message $\mathbf{m}$ into the result.

The encrypted packet is then generated by the following function:

$$\mathbf{packet} \leftarrow \texttt{encryption}(\mathbf{m}, \mathbf{A}, \mathbf{Y})$$

Algorithm 7: `encryption`

```
Input: $\mathbf{m} \in \mathcal{C}$, $\mathbf{A} \in \mathbb{Z}_2^{k\times n}$, $\mathbf{Y} \in \mathbb{Z}_2^{l\times n}$
Output: $\mathbf{enc} \in \mathbb{Z}_2^{k+l}$
$\mathbf{e} \leftarrow$ weighted_array($n, t$)
$\mathbf{nnc} \leftarrow$ row_column($\mathbf{A}, \mathbf{e}$)
$\mathbf{cmp} \leftarrow$ row_column($\mathbf{Y}, \mathbf{e}$)
$\mathbf{cmp} \leftarrow$ sum_array($\mathbf{cmp}, \mathbf{m}$)
$\mathbf{enc} \leftarrow$ concat($\mathbf{nnc}, \mathbf{cmp}$)          /* Concatenate the two arrays */
return $\mathbf{enc}$;
```

[1]A nonce is a unique, one-time-use value used to ensure secure communication.

2.2.4 Decryption

Upon receiving an encrypted message, the nonce is separated from the message, and the first part is utilized to compute the decryption key. Specifically, the private key $\mathbf{S}$ is applied as follows:

$$\mathbf{SAe}^T \leftarrow \texttt{row\_column}(\mathbf{S}, \mathbf{nnc})$$

This result is used as the decryption key, with the vector $\mathbf{key} = \mathbf{SAe}^T$, applied as follows:

$$\mathbf{key} = \mathbf{Ye}^T - \mathbf{Ee}^T.$$

Finally, the original message is retrieved by subtracting this key from the received code:

$$\mathbf{cmp} - \mathbf{key} = \mathbf{m} + \mathbf{Ye}^T - \mathbf{Ye}^T + \mathbf{Ee}^T$$

which simplifies to recover the message:

$$\mathbf{message} \leftarrow \texttt{decryption}(\mathbf{packet}, \mathbf{S}).$$

Algorithm 8: `decryption`

```
  Input: $\mathbf{packet} \in \mathbb{Z}_2^{k+l}$, $\mathbf{S} \in \mathbb{Z}_2^{l \times n}$
  Output: $\mathbf{dec} \in \mathbb{Z}_2^{l}$
1 $\mathbf{key} \leftarrow$ row_column($\mathbf{S}, \mathbf{packet}[..k]$)      /* Sub-vector from 0 to k-1 */
2 $\mathbf{dec} \leftarrow$ sub_array($\mathbf{packet}[k..], \mathbf{key}$)    /* Sub-vector from k to end */
3 return $\mathbf{dec}$;
```

Although it might seem that this operation yields the decrypted message, the presence of the term $\mathbf{Ee}^T$ introduces an error, which we address in the subsequent section on error correction.

2.2.5 Error Correction

At this stage, as previously mentioned, the message is represented as $\mathbf{m} + \mathbf{Ee}^T$. Despite the added error, the message $\mathbf{m}$ is equipped with error correction, allowing for the recovery of the original message despite the presence of $\mathbf{Ee}^T$.

One practical method is to send the same message multiple times, using different nonces, and then apply majority voting to determine the most likely correct bits after decryption.

Chapter 3

Future Developments

Public-key cryptography employs a pair of keys: a public key, which is accessible to everyone, and a private key, kept confidential. The public key allows anyone to encrypt messages, but only the holder of the private key can decrypt them. This approach simplifies secure communication by eliminating the need for prior secure key exchange, allowing secure interactions through the exchange of public keys.

Quantum computing introduces qubits, which can exist in multiple states simultaneously due to the phenomenon of superposition. This property, combined with entanglement, significantly enhances the computational power of quantum systems. The Quantum Fourier Transform (QFT), a quantum analog of the classical Fourier transform, is particularly important in quantum algorithms for identifying periodicity in superpositions.

RSA cryptography relies on the difficulty of factoring large composite numbers. Specifically, RSA security is based on the challenge of determining the prime factors of a large number N, which is the product of two prime numbers. The RSA algorithm uses these prime factors to generate public and private keys, with the system's security depending on the infeasibility of factorizing N within a reasonable time frame using classical computers.

While classical factorization methods, such as the General Number Field Sieve algorithm, exist, their practical application is limited by significant computational requirements. Quantum computers, however, can execute Shor's algorithm, which efficiently solves the integer factorization problem. The Quantum Fourier Transform is integral to this process, as it identifies the periodicity in functions corresponding to the factors of N, thus enabling the determination of the private key.

Given the rapid advancements in quantum computing and the decreasing qubit requirements for practical quantum algorithms, there is an urgent need to develop post-quantum cryptographic solutions. These solutions aim to secure digital communications against the capabilities of quantum adversaries, ensuring data integrity and confidentiality in a post-quantum world. The development and standardization of such cryptographic methods are critical to maintaining robust security frameworks in the face of emerging quantum threats.

Bibliography

[1] Michael Alekhnovich. More on average case vs approximation complexity. *Comput. Complex.*, 20(4):755–786, 2011.

[2] Daniel J. Bernstein. librandombytes: Cryptographically secure random number generator. `https://randombytes.cr.yp.to/index.html`, 2023.

[3] David Blackman and Sebastiano Vigna. Scrambled linear pseudorandom number generators. *ACM Trans. Math. Softw.*, 47(4):36:1–36:32, 2021.