POLITECNICO
MILANO 1863

Alekhnovich's Cryptosystem

PII

Course Instructor: Ing. Alessandro Barenghi

August 30, 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 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.

1.1.2 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,
$$C = A \times B = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 41 & 50 \end{bmatrix}.$$

1.1.3 Kernel of a Matrix

A key concept related to matrices and linear transformations in vector spaces is the **kernel**. The kernel provides insight into the solutions of linear systems and the properties of the transformations defined by matrices.

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.

Understanding the kernel is crucial in many areas, including coding theory and cryptography, where it plays a role in defining codes and analyzing their properties.

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 $\mathcal{C}$.

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.

Computational Complexity Understanding the security of cryptographic systems requires a grasp of computational complexity, particularly the classification of problems based on the time it takes to solve them relative to the size of the input. Problems that can be solved in polynomial time, denoted as P, are considered feasible because the time required grows at a manageable rate as the input size increases. However, many problems, including those crucial to cryptography, belong to a class called NP (nondeterministic polynomial time), where verifying a given solution is feasible, but finding the solution may not be. NP-complete problems, a subset of NP, are particularly significant because they are the hardest problems in NP; solving any NP-complete problem efficiently would imply that all NP problems could also be solved efficiently.

Decisional Decoding Hypothesis The Decisional Decoding Hypothesis posits that no polynomial-time decoding algorithm can distinguish between a random vector and a vector that is a codeword corrupted by noise of a specific weight, given a random code defined by either a uniformly random generating matrix or a uniform random parity-check matrix. This hypothesis plays a crucial role in assessing the security of certain cryptographic systems, ensuring that distinguishing between random noise and encoded messages is computationally infeasible.

DDH. The decisional decoding hypothesis assumes that for parameters t, k, and n within bounds $0 < R_1 \leq \frac{k}{n} \leq R_2 < 1$, no polynomial-time algorithm A can distinguish with non-negligible advantage between:

- **Case 1:** A uniformly random vector $\mathbf{u} \in \mathbb{F}_2^n$.
- **Case 2:** A vector $\mathbf{c} + \mathbf{e}$, where $\mathbf{c}$ is a random codeword from a linear code C and $\mathbf{e}$ is a noise vector of Hamming weight t.

Chapter 2

Alekhnovich's Cryptosystem

This chapter provides an overview of Alekhnovich's cryptosystem, focusing on its foundational principles and the key operations involved. Although the discussion does not delve deeply into the algorithm's potential advantages against quantum computers compared to existing algorithms, it aims to provide sufficient context for subsequent sections.

2.1 The Algorithm

The Alekhnovich cryptosystem [1] is based on hard problems in coding theory, particularly the challenge of decoding random linear codes, a problem believed to be difficult for both classical and quantum computers.

2.1.1 Keys generation

The cryptosystem operates by first generating two random matrices: $\mathbf{A} \in \mathbb{Z}_2^{k\times n}$ and $\mathbf{S} \in \mathbb{Z}_2^{l\times k}$. Here, $\mathbf{A}$ serves as the generator matrix for a random linear code, while $\mathbf{S}$ is a scrambling matrix that transforms the code generated by $\mathbf{A}$.

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

The public key 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 critical as it enables the recovery of the message even if it has been corrupted by noise.

2.1.2 Encryption and Decryption Process

The encryption and decryption processes of the Alekhnovich cryptosystem are outlined 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 plaintext 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, $\mathbf{Ae}^T$, by the secret key $\mathbf{S}$ to obtain $\mathbf{SAe}^T = \mathbf{Ye}^T - \mathbf{Ee}^T$.
2. The receiver then subtracts this result from the second part of the encrypted message, $\mathbf{m}+\mathbf{Ye}^T$, yielding:

$$(\mathbf{m} + \mathbf{Ye}^T) - (\mathbf{Ye}^T - \mathbf{Ee}^T) = \mathbf{m} + \mathbf{Ee}^T.$$

3. Given that $\mathbf{E}$ is sparse, the vector $\mathbf{Ee}^T$ is expected to have a low weight, and the error-correcting code $\mathcal{C}$ is then used to recover the original message $\mathbf{m}$ from $\mathbf{m} + \mathbf{Ee}^T$.

2.2 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 over determined, 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: $l = 13000$ *bits,* $t = 114$ *bits,* $k = 1300$ *bits, and* $n = 16000$ *bits.*

The choice of k as approximately 10% of l is based on an initial heuristic aimed at balancing computational efficiency with security. This ratio ensures that the scrambling matrix's complexity remains manageable while keeping the system over determined ($k+l < Rn$, where $R < 1$), a critical condition for security. Additionally, the error weight t is set high enough to ensure the decoding problem remains computationally intractable. These parameter choices, especially the relationship between k, l, and n, will be refined through experimentation and security analysis to optimize performance and robustness.

2.3 Security Considerations

The security of Alekhnovich's cryptosystems is fundamentally reliant on the **Decisional Decoding Hypothesis (DDH)**. But to comprehensively evaluate the security implications, it is essential to examine the relationship between DDH and the Search Decoding Problem (SDP), which is at the core of decryption.

Search Decoding Problem In cryptography, the Search Decoding Problem involves identifying the original codeword from a noisy version received through a communication channel. This problem is critical for decoding encrypted messages, particularly when the noise pattern is predefined and the code structure is known. The intractability of the SDP is assumed to ensure that without the private key, decrypting the message remains computationally unfeasible.

2.3.1 Reduction to the Search Decoding Problem

Alekhnovich's cryptosystem relies on the assumption that the SDP is intractable under the given parameters. This intractability is crucial for ensuring that an adversary cannot decrypt the message without the private key. The importance of the SDP difficulty is evident in the following paragraphs:

Deciphering a Message Without the Secret Key In Alekhnovich's cryptosystem, decoding a message without the secret key effectively means solving the SDP. The challenge is to determine the original codeword $\mathbf{c}$ from its noisy version $\mathbf{c}+\mathbf{e}$, where $\mathbf{e}$ is an error vector of known weight t. The difficulty of this task is what prevents an adversary from decrypting the message without access to the private key. The public key, usually represented by a transformation matrix $\mathbf{G}$, is used for encoding the message through linear transformations and the addition of noise. An attacker who could solve the SDP would be able to invert this operation, directly accessing the original message $\mathbf{m}$ from the transmission $\mathbf{xG}+\mathbf{e}$.

Successfully reconstructing $\mathbf{c}$ *from* $\mathbf{c}+\mathbf{e}$ *implies a solution to the SDP, which would challenge the Decisional Decoding Hypothesis with parameter* t*. This hypothesis posits that the noisy codeword* $\mathbf{c}+\mathbf{e}$ *is computationally indistinguishable from a uniformly random vector, thereby securing the encryption.*

Recovering the Private Key Recovering the private key from publicly available data would require exploiting potential weaknesses in the structure of the encoding scheme used to generate G. If an attacker could reverse-engineer the transformations encoded by $\mathbf{G}$, or infer its effects (i.e., deducing $\mathbf{E}$ or the structure of $\mathcal{C}$ from $\mathbf{G}$), they could decrypt any subsequent messages, rendering the private key unnecessary.

The ability to decrypt messages without the private key or extract the key from public components would fundamentally undermine the security model. This would violate the Decisional Decoding Hypothesis, as it would imply that encoded messages are distinguishable from random vectors, making deterministic decryption computationally feasible.

2.3.2 Complexity of the Search Decoding Problem

A reduction to an NP problem involves transforming one problem into another in such a way that solving the new problem efficiently would also solve any problem in NP, highlighting the original problem's inherent computational difficulty.

SDP. Given the Decisional Decoding Hypothesis (DDH), it is postulated that no efficient algorithm exists that can solve the Search Decoding Problem (SDP). Specifically, the objective is to recover the codeword $\mathbf{c}$ from its noisy version $\mathbf{c} + \mathbf{e}$, despite knowing the structure of the code $\mathcal{C}$ and the parameters k and n.

Proof. To illustrate the reduction:

1. Assume there exists a polynomial-time algorithm B capable of solving the SDP by recovering $\mathbf{c}$ from $\mathbf{c} + \mathbf{e}$.

2. Construct an auxiliary algorithm A that employs B to attempt decoding an input vector $\mathbf{v}$, hypothesizing it to be $\mathbf{c} + \mathbf{e}$.

3. If B successfully decodes $\mathbf{v}$ to a codeword $\mathbf{c}$ and $\mathbf{v} - \mathbf{c}$ matches the noise pattern of weight t, then A outputs "noisy codeword"; otherwise, it outputs "random vector".

This approach contradicts the Decisional Decoding Hypothesis by providing a means to distinguish between noisy codewords and random vectors, thereby affirming the complexity of the SDP under the DDH assumption. □

2.3.3 Practical Implications

The intractability of the SDP and the strength of the DDH are not just theoretical concerns—they have significant practical implications for the security of Alekhnovich's cryptosystem. In practical cryptographic implementations, the parameters t, k, and n must be carefully chosen to ensure that the SDP remains infeasible for any potential adversary. If these parameters are poorly chosen, or if advancements in algorithms or computing power reduce the difficulty of the SDP, the security of the entire system could be compromised.

Additionally, this security model relies heavily on the assumption that e, the noise vector, is truly random and of the prescribed weight t. Any predictability or deviation from this assumption could provide attackers with the leverage needed to challenge the DDH and, by extension, the cryptosystem's security.

Chapter 3

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.

3.1 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.

3.1.1 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 */
```

3.1.2 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.

3.1.3 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: $m = n \times \text{sizeof}$; $p = \frac{l}{\text{sizeof}}$
  $\mathbf{T} \leftarrow$ 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 $+ k$
              $w \leftarrow \left\lfloor \frac{i}{\texttt{sizeof}} \right\rfloor$
              $s \leftarrow \texttt{sizeof} - (i \bmod \texttt{sizeof}) - 1$
              $t_{yw} \leftarrow t_{yw}$ | shift_bit(bit, $s$)
  return $\mathbf{T}$
```

3.1.4 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 $result$;
```

3.2 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.

3.2.1 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 ∈ $\mathbb{N}$
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.

3.2.2 Matrix S

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

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

3.2.3 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 ∈ $\mathbb{N}$
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     $a_i = a_i \mid \texttt{shift\_bit}(1, s)$          /* Set the bit at calculated position */
10 return $\mathbf{a}$
```

3.2.4 Matrix Y

To compute matrix $\mathbf{Y}$, we need to first transpose matrix $\mathbf{A}$ so that its rows become columns. Then, we perform a bitwise row-by-column multiplication between the transposed matrix $\mathbf{A}^T$ and matrix $\mathbf{S}$. Finally, we add matrix $\mathbf{E}$ to the resulting matrix to obtain the final matrix $\mathbf{Y}$. 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: $\mathbf{Y} \in \mathbb{Z}_2^{l\times n}$
1 $\mathbf{T} \leftarrow$ matrix_transposed($\mathbf{A}$)
2 for $i$ in range($l$):
3     for $j$ in range($n$):
4         $y_{ij} \leftarrow$ bax($\mathbf{S}_j, \mathbf{T}_i$)        /* Bit-wise row-by-column product using BAX */
5 for $i$ in range($l$):
6     for $j$ in range($n$):
7         $y_{ij} \leftarrow y_{ij}$ | $e_{ij}$
8 return $\mathbf{Y}$
```

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

2. $\mathbf{cmp} \leftarrow$ `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: $\mathbf{enc} \in \mathbb{Z}_2^{k+l}$
1 $\mathbf{e} \leftarrow$ weighted_array($n, t$)
2 $\mathbf{nnc} \leftarrow$ row_column($\mathbf{A}, \mathbf{e}$)
3 $\mathbf{cmp} \leftarrow$ row_column($\mathbf{Y}, \mathbf{e}$)
4 $\mathbf{cmp} \leftarrow$ sum_array($\mathbf{cmp}, \mathbf{m}$)
5 $\mathbf{enc} \leftarrow$ concat($\mathbf{nnc}, \mathbf{cmp}$)                    /* Concatenate the two arrays */
6 return $\mathbf{enc}$;
```

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

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

3.5 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 4

Project Documentation

This chapter provides an overview of the C files included in the project. The files documented here are:

- `test.c`
- `backend.c`
- `api.c`
- `main.c`

Each file plays a specific role in the overall project structure, and this chapter will detail their functionality.

4.1 `main.c`

The `main.c` file serves as the entry point for the program. It processes command-line arguments to determine which operation the program should perform, such as testing, key generation, encryption, decryption, or error correction.

4.1.1 Key Functions

- `main(int argc, char *argv[])`: It parses the command-line arguments, determines the appropriate command to execute, and invokes the corresponding function.
- `Command get_command(const char *command)`: It maps a command string to its corresponding enumeration value.
- `void print_err(const char *name, const char *msg)`

4.1.2 Details

The `main.c` file handles the main logic of the application by evaluating the command-line input and executing the appropriate operation based on the command provided. The supported commands include:

- **TEST**: Runs the test suite by invoking the `test()` function from the `test.c` file.

- **GENERATE**: Calls the `generate_key()` function to generate encryption keys.
- **ENCRYPT**: Encrypts a message using the provided key paths, handled by the `encrypt()` function.
- **DECRYPT**: Decrypts a message using the provided file paths, handled by the `decrypt()` function.
- **CORRECT**: Corrects data using input files, handled by the `correct()` function.

The control flow in the `main()` function is structured as follows:

1. Argument Validation: checks if at least one command-line argument is provided.
2. Command Identification: is used to identify the command type based on the first argument.
3. Command Execution: the program invokes the appropriate function. If insufficient arguments are provided for a specific command, an error message is printed using `print_err()`.
4. Error Handling: if the command is invalid, an error message is displayed, and the program exits with a status code.

The `main.c` file integrates closely with other modules like `api.c` and `test.c`, enabling it to act as a central coordinator for different operations in the application. The use of an enumeration for commands makes it easy to extend the program with additional functionalities in the future.

4.2 `test.c`

The `test.c` file contains two key functions for testing the cryptographic processes: `test()` and `shortcut()`. These functions serve different purposes and offer distinct levels of testing and validation.

4.2.1 `test() Function`

The `test()` function is the primary testing function that performs a comprehensive validation of the cryptographic operations. It is invoked with an integer parameter `mod` to control its behavior.

- Comprehensive Testing: When `mod` is set to '0', the `test()` function conducts a full testing process, which includes generating random matrices, writing and reading keys, encrypting and decrypting a random message, and verifying the integrity of the entire process.
- Matrix Operations: The function generates matrices required for encryption and decryption, ensuring that the cryptographic algorithms work correctly with dynamically generated data.
- Resource Management: It manages resources meticulously, allocating and freeing memory for matrices and arrays to prevent memory leaks.
- Validation: The function checks that the generated keys can be correctly read back from the files and that the encryption and decryption processes produce the expected results.

The `test()` function is designed for a deep and thorough validation of the entire cryptographic process. It is used to ensure that all components work together as expected under typical conditions.

4.2.2 `shortcut()` Function

The `shortcut()` function provides a more streamlined testing approach compared to `test()`. It is designed for quick validation and includes additional steps for error correction.

- Key Reading: Unlike `test()`, the `shortcut()` function skips matrix generation and directly reads pre-existing keys from files. This makes the process faster but assumes that the keys have already been correctly generated and stored.
- Encryption and Decryption: The function encrypts and decrypts a random message using the pre-existing keys, similar to the `test()` function, but with less emphasis on the full lifecycle of key management.
- Error Correction: A significant difference is that `shortcut()` includes an error correction phase. It encrypts the message three times, reads the resulting packets, and attempts to correct any errors that may have occurred during transmission.
- Efficiency: The `shortcut()` function is optimized for speed and efficiency, making it ideal for situations where a quick test is needed, rather than a full validation.

The `shortcut()` function is useful for quickly verifying that the core cryptographic operations are functioning as intended, particularly in scenarios where error correction is critical. However, it does not perform the full range of checks that the `test()` function does.

4.3 `api.c`

The `api.c` file provides the core API functions for the cryptographic system, including key generation, encryption, decryption, and error correction. These functions are responsible for the main cryptographic operations and serve as the interface between the higher-level application logic and the lower-level data handling provided by the backend.

4.3.1 Key Functions

`generate_key()` generates the cryptographic keys necessary for the encryption and decryption processes. It creates three matrices: `a` (public key matrix), `s` (private key matrix), and `e` (error matrix). The function then computes a fourth matrix, `y`, which is derived from `a`, `s`, and `e`. These keys are then written to files using the backend's `write_key()` function.

`encrypt(const char *mex, const char *a_path, const char *y_path)` handles the encryption of a message. It reads the public keys `a` and `y` from the specified file paths and then encrypts the provided message `mex`. The encryption process involves generating a random error array `e`, calculating the `nnc` and `word` arrays through matrix multiplication and XOR operations, and then writing these arrays to files using the `write_packet()` function.

`decrypt(const char *fnnc, const char *fword, const char *key_path)` decrypts a message by reversing the encryption process. It reads the private key `s`, as well as the `nnc` and `word` arrays from the specified file paths. The decryption involves matrix multiplication and XOR operations to retrieve the original message. The decrypted message is then written to a file.

`correct(const char *path1, const char *path2, const char *path3)` performs error correction on three encrypted data packets. It reads the three packets from the specified paths and applies an error correction algorithm using the `correct_errors()` function from the backend. The corrected data is then written to a file.

4.4 `backend.c`

The `backend.c` file provides essential input/output operations for handling cryptographic data structures, such as matrices and arrays. It includes functions for reading and writing keys, converting data formats, and correcting errors in transmitted packets. This file acts as the backbone of data handling within the cryptographic system, ensuring that data is correctly stored, retrieved, and processed.

4.4.1 Key Functions

`write_key(const char *path, struct mat *m)` writes a matrix (used as a cryptographic key) to a binary file. It stores the matrix's dimensions (rows and columns) followed by its data, row by row.

`read_key(const char *path)` reads a matrix from a binary file and reconstructs it into memory. It is the counterpart to `write_key()`, enabling the cryptographic system to retrieve previously stored keys and use them for operations such as encryption, decryption, and validation.

`convert_to_array(const char *filepath)` converts a text file of binary data (represented as '1's and '0's) into an array of 64-bit unsigned integers. This conversion is essential for preparing data in the correct format for encryption or other cryptographic processes. The resulting array can be used directly in cryptographic operations, ensuring efficient processing of binary data.

`write_packet(const char *output_path, struct arr *message)` writes a data packet (an array of data) to a binary file. It first writes the length of the array and then the data itself. This function is used to store encrypted messages or any other arrays that need to be transmitted or stored for later use.

`read_packet(const char *path)` reads a data packet from a binary file, reconstructing the array in memory. It is typically used to retrieve encrypted messages or other data structures that were previously stored using `write_packet()`.

`correct_errors(struct arr *a, struct arr *b, struct arr *c)` performs error correction on three arrays by applying a bitwise operation to correct errors that may have occurred during transmission. It combines the data from three sources and resolves differences to produce a corrected array.

4.5 Launching the Program

To launch the program, follow these steps. The process involves setting up the required dependencies, building the project using CMake, and running the compiled executable.

4.5.1 Install `librandombytes`

The program depends on the `librandombytes` library, which needs to be downloaded and installed before building the project from the source alreay present in the paper.

4.5.2 Build the Project using CMake

Once `librandombytes` is installed, you can build the project using CMake. The `CMakeLists.txt` file provided in the project directory contains all the necessary instructions.

- Navigate to the root directory of the project.
- Create a build directory and navigate into it:

  ```
  $ mkdir build
  $ cd build
  ```

- Run CMake to generate the makefiles:

  ```
  $ cmake ..
  ```

- Build the project using make:

  ```
  $ make
  ```

4.5.3 Run the Program

After successfully building the project, you can run the compiled executable.

- The executable will be located in the `build/bin` directory. To run the program, use the following command:

  ```
  $ ./your_program_name <command> [<args>]
  ```

- Replace `your_program_name` with the actual name of the executable generated by the build process.
- The available commands are those described in the `main.c` section.

4.5.4 Verifying the Setup

To ensure everything is set up correctly, you can run a test command such as:

```
$ ./your_program_name test
```

This command will initiate the testing process, verifying that the encryption, decryption, and key generation functionalities are working as expected.

Chapter 5

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.