POLITECNICO
MILANO 1863

Alekhnovich's Cryptosystem

PII

Course Instructor: Ing. Alessandro Barenghi

April 3, 2024

Introduction

In 2006, Clive Humby coined a slogan that has proven increasingly prescient over time: "Data is the new oil." Nothing could be truer in today's context. Much like any valuable resource, data has become a coveted asset, with individuals and entities vying to acquire and leverage it to their advantage. Consequently, from the outset, the need for secure communications and data protection was paramount.

Over the years, technological advancements have given rise to new algorithms and protocols designed to maximize data security. Tremendous strides have been made in safeguarding sensitive information, and yet, we find ourselves facing an imminent threat: the advent of quantum computers.

This threat to data security is not a hypothetical scenario; it is a reality on the horizon. Quantum computers, leveraging the principles of quantum mechanics, possess the potential to reshape the digital landscape by solving complex problems at unprecedented speeds. As we increasingly rely on digital communication and data storage for our most critical operations, the need for robust cryptographic defenses against quantum threats has never been more urgent.

Contents

Chapter 1

THEORETICAL FUNDAMENTALS

To address the imminent threat 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 undoubtedly essential to introduce several concepts from the fields of coding theory and 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, delves into the analysis of vector spaces and their transformations. Its significance lies in its wide-ranging applications, encompassing the solution of systems of linear equations and the exploration of multidimensional objects in space. To undertake these analyses, linear algebra employs a diverse toolkit, including vector spaces, linear equations, and matrices.

Matrices, rectangular arrays of numbers, symbols, or expressions meticulously organized in rows and columns, assume a pivotal position in linear algebra. They offer a concise and structured representation of vectors, linear transformations, and systems of linear equations. Matrices, amenable to various operations such as addition, multiplication, and transposition, open pathways to tackling a broad spectrum of mathematical challenges

1.1.1 Matrices Sum

The sum of two matrices A and B of the same dimension is defined as the matrix C whose entries are the sum of the corresponding entries of A and B. In other words, if $A = [a_{ij}]$ and $B = [b_{ij}]$, then $C = [c_{ij}]$ where $c_{ij} = a_{ij} + b_{ij}$.

1.1.2 Matrices Product

The product of a matrix A with dimensions $(m \times n)$ and a matrix B with dimensions $(n \times p)$ is defined as a matrix C with dimensions $(m \times p)$. The entry c_{ij} of C is defined as the sum of the

products of the entries in row i of A and the corresponding entries in column j of B. In other words, $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$

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

1.1.3 $\mathbb{Z}_2$ Field and $\mathbb{Z}_2^n$ Vector Space

In both cases, we refer to the set $\mathbb{Z}_2$, representing the set of integers mod 2, euqals to $\{0, 1\}$. In the case of fields, this set combines two operations that remain closed concerning the field set. Regarding the vector space $\mathbb{Z}_2^n$, its made by vectors of size n, where each element belongs to $\mathbb{Z}_2$, where addition and scalar multiplication are defined operations, also closed within the set. Furthermore, there is certainty about the validity of certain properties such as associativity and distributivity.

In summary, a field constitutes a structure with specified arithmetic properties, while a vector space denotes a collection of objects (vectors) adhering to specific regulations (dictated by vector addition and scalar multiplication), and these vectors can be formed over a field. Fields furnish the scalars necessary in defining vector spaces.

1.2 Coding Theory

The term 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, images, etc., into other forms that are more suitable for storage, transmission, or encryption.

1.2.1 Code

The code it can seen as a function that given a word x encode that word over a chosen alphabet as $C : X \to \sum^*$, where $C(x)$ is the code associated whith x. Proprietis of code:

1. injective $\to$ non-singular(the length is arbitrary)
2. non-singular $\to$ uniquely decodable

1.2.2 Hamming Distance

The Hamming distance is a measure used to quantify the dissimilarity between two strings of equal length in information theory, coding theory, and computer science. 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.

1.2.3 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. Various algorithms have been implemented over time, ranging from the most obvious and simple to the most effective and complex, including:

- 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.
- Error-correcting Code: this represents more of an error detection family. A code with a minimum Hamming distance can identify up to $d-1$ errors in a code word (e.g. $d=2 \rightarrow$ Parity bit, $d=4 \rightarrow$ Extended Hamming).

1.2.4 Generator and Check Matrices

In the context where the code C represents a linear subspace of $\mathbb{F}_q^n$, it can be expressed as a linear combination of its codewords, which form a basis typically organized into rows within a generating matrix G. This matrix serves as a representation of the code itself, simplifying the transformation of data vectors into code words through straightforward matrix multiplication. Subsequently, a check matrix (or parity check matrix) is constructed, characterized by its kernel [1] being C. The code that H can generate is termed the dual code of C. When the G matrix takes the form $[\mathbb{I}_k|P]$, it is considered to be in standard form, and consequently, the H matrix appears as $[-P^T|\mathbb{I}_{n-k}]$.

1.2.5 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.

[1] Which is typically the pre-image of 0. An important special case is the kernel of a linear map. The kernel of a matrix, also known as the null space, refers to the kernel of the linear map defined by the matrix.

Chapter 2

Alekhnovich's Cryptosystem

2.1 Overview

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.

2.1.1 The Algorithm

The algorithm begins by generating two random matrices, $A \in \mathbb{Z}_2^{k \times n}$ and $S \in \mathbb{Z}_2^{l \times k}$. After, it generates a third random matrix, $E \in \mathbb{Z}_2^{l \times n}$. The rows of E are chosen from random and independent vectors of weight t (the weight is computed as $\sqrt{n}$).The algorithm then defines the matrix Y as $Y = SA + E$.

The matrices Y and A represent the public key of the algorithm. The public key is shared with anyone who wants to send encrypted messages to the sender. The plain text to encrypt space is $m = \mathbf{C} \subset \{0,1\}^l$ where $\mathbf{C}$ is an error-correcting code. The error-correcting code implies an algorithm used to ensure that the message can be recovered even if it is corrupted by noise. Now following steps are required for a secure exchange of information:

- To encrypt the message, the sender performs the following steps:

  1. The sender generates a random t-weight vector e of $\mathbf{F}_2^n$.
  2. The sender multiplies e^T by the public key Y and A.
  3. The sender adds the message m to the result of the multiplication Ye^T.

  The encrypted message is then given by the following vector:

  $$C(m) = (Ae^T, m + Ye^T)$$

  .

- To decrypt the message, the receiver performs the following steps:

  1. The receiver multiplies the first part of the encrypted message, Ae^T, by the secret key S getting $SAe^T = Ye^T - Ee^T$.
  2. The receiver subtracts from the second part of the encrypted message, $m + Ye^T$, the result of the previous multiplication.

Now the receiver finds itself with a vector of the type:

$$(m + Ye^T) - (Ye^T - Ee^T) = m + Ee^T$$

The algorithm works by exploiting the fact that the product of the matrix E and the vector e of weight t is likely to be zero. This is due to the random positions of the few 1s in both both the rows of E and the vector e; the probability that even just one of these coincides is very low. In the worst case, the code will distance itself from the original with a maximum Hamming distance of t, which is manageable with the error-correcting algorithm of the chosen code.

2.1.2 Parameters

The algorithm's implementation requires defining several parameters, including matrix dimensions and message length. To ensure the cryptosystem's reliability, we must impose a condition on the matrix dimensions:

$$k + l < Rn$$

where R is a constant less than 1. Once the message length (l bits) is chosen, this rule guides the selection of suitable values for k and n.

An important implementation factor is the practical limitation of allocating matrices with single-bit cells. A common solution is to select matrix dimensions that are divisible by the size of standard integers in the chosen programming language (e.g., *uint64_t* in C).

Additionally, remember:

- The weight of the vector t is
$$t = \sqrt{l}$$
- The transmission channel's error probability is
$$p = \frac{t^2}{n}$$

For our implementation, we'll use the following parameters value: $l = 1300$ *bits,* $t = 144$ *bits,* $k = 2^6$ *bits,* $n = 2^{17}$ *bits.*

2.2 Key Generation

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'll employ the following libraries for robust randomness:

- **librandombytes:** generates cryptography secure random seeds (`randombytes(x, X_BYTES)` API, see here).
- **xoshiro256 PRNG:** efficiently generates pseudo-random bits (look here for the documentation).

2.2.1 Matrix A

Following the referenced work, matrix A has dimensions $k \times n$ and contains random values. We'll delegate random value generation to trusted functions. To create A a simple function will be used which, after initializing the seed, will generate as many random numbers as necessary:

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

Note: If n represents the total number of bits, we can reduce A's size: $a.columns \leftarrow \frac{n}{s}$, where s is the *size-of* the integer type chosen for our matrix.

Algorithm 1: `random_matrix`

Input: rows, columns $\in \mathbb{N}$
Output: random matrix $\in \mathbb{N}^{r \times c}$

```
1 init_seed()                                   /* Initializing the seed */
2 M ← matrix
3 for i in range(rows):
4     for j in range(columns):
5         m_ij ← xoshiro256.next()                /* Filling the matrix */
6 return M
```

2.2.2 Matrix S

We follow the exact same procedure for matrix S, simply changing the input variables to our function:

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

As before, we reduce the column size to the ratio of the number of bits needed to the size of the instantiated matrix type.

2.2.3 Matrix E

This matrix has different requirements than the previous ones: a predetermined number of 1-bits (defined as weight t) must be randomly positioned within each row. To achieve this, we'll utilize a function to generate `e.rows` arrays, each containing a constant number of 1s, through a loop:

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

Since we are addressing random positions within individual bits of an integer vector, it is pertinent to emphasize the processing of these integers that will form rows of the matrix:

Algorithm 2: `weighted_array`

Input: l, t $\in \mathbb{N}$
Output: random weighted vector $\in \mathbb{N}^l$

```
a ← array
for count in range(t):
    repeat
        p ← xoshiro256.next()
        i ← p divided the size of the array's type
        s ← p module the size of the array's type
    until no 1 has been inserted at that position yet
    a_i ← OR with a 1 shifted by s bits      /* Set the bit at calculated position */
return a
```

2.2.4 Matrix Y

Matrix Y, in conjunction with the precomputed A matrix, constitutes the crucial public key for encrypting messages. The computation process involves:

1. Multiplying the matrices S and A.
2. Adding the matrix E to the previously obtained result.

$$Y \leftarrow \texttt{compute\_pub(S, A, E)}$$

Apart from the straightforward addition operation between matrices, it's essential to delve into the row-column product, especially when dealing with matrices where the fundamental elements are individual bits of integers rather than entire integer cells.

Transposition Initially, we must devise an efficient strategy for selecting the columns of bits in matrix A. One approach involves transposing the individual bits of the matrix, effectively converting its rows into columns for subsequent computations.

Algorithm 3: `matrix_transposed`

Input: M $\in \mathbb{N}^{k \times n}$
Output: T $\in \mathbb{N}^{n \times h}$
Data: by *sizeof* we mean the type cell of the matrix

```
T ← matrix                                  /* of n rows, and k / size-of columns */
for i in range(m.rows):
    for j in range(m.columns):
        for k in range(sizeof):
            bit ← fetch_bit(m_ij)                          /* select the k^th bit */
            t_yw ← bit shifted by i (mod sizeof)     /* y = (j * 64) + k,  w = i / size-of */
return T
```

Bit-wise AND & XOR Contextualizing the previous chapter, traditional arithmetic operations such as multiplication and addition are not applicable here for the row-by-column product. Instead, we perform bitwise operations on matrices. Initially, we conduct a bitwise AND operation between

the cells of the two matrices, followed by an XOR operation on the results of these AND operations, as described below:

Algorithm 4: `bax`

Input: $\mathbf{a}, \mathbf{v} \in \mathbb{N}^l$
Output: result $\in \mathbb{N}_2$

```
result ← 0
for i in range(l):
    and ← a_i&v_i                          /* bit-wise AND operation */
    result ← result + count_ones(and)
result ← result (mod 2)                    /* simplified XOR */
return result;
```

where the **a** and **v** arrays represent the S and A^T single row.

2.3 Encryption

Following the generation of cryptographic keys, they are utilized for encryption and decryption purposes. In the transmission of encrypted messages, apart from selecting an appropriate element from our cryptographic system, denoted as $m = \mathbf{C} \subset \{0,1\}^l$, it is imperative to generate a random weighted vector, maintaining the previous weight t. This can be achieved through the function introduced earlier:

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

For the encryption procedure itself, we can rely on the previously introduced functions. A call to two functions is sufficient:

1. **nonce** ← `compute_nonce(A, e)`, computing the row-by-column product (with a "matrix" consisting of a single column)
2. **code** ← `encipher(Y, e, m)`, which, in addition to performing the row-by-column product like the one above, also adds our message m to the result

2.4 Decryption

2.5 Error Code

Bibliography

[1]

[2]

[3]

[4]

[5]