Imperial Project Report
Cryptography: From Caesar to Quantum

Tom Ballantyne, Edward Ștefan Bujdei, Amit Paul, Eden Zan

November 2024

Contents

1 Introduction

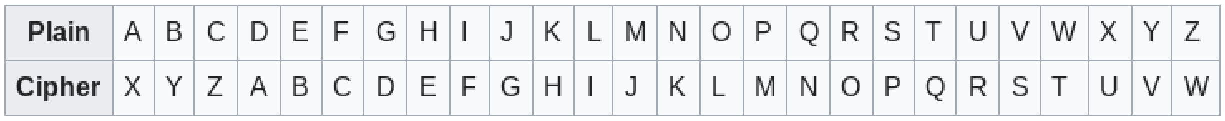
Encryption methods have existed for thousands of years in order to safely transmit messages from one party to another without 3rd parties intercepting the data. These vary from rudimentarily designating each letter in the alphabet with another letter, and replacing each character in the plaintext with its designated letter, a class of ciphers known as *monoalphabetic substitution ciphers*, to more complex algorithms that exploit the hardness of prime factorisation, for example in the RSA cipher, to ensure the hardness of decrypting the ciphertext. However, modern encryption methods including the RSA cipher, upon which all of cybersecurity depends on, have been threatened by known algorithms designed to run on quantum computers, and while these require a large amount of computational resources to be efficient, recent and potential future developments in quantum computing have and will further put these ciphers at risk. We have elected to research and document candidate future post-quantum resistant ciphers, ciphers designed to run on classical computers but be immune to quantum algorithms, suggested by NIST, the National Institute of Standards and Technology, to assess the safety and reliability of future communication

1.1 Caesar Cipher

One early encryption method was the *Caesar Cipher*, a monoalphabetic substitution cipher used by Julius Caesar to securely send messages, which can be generalised using modular arithmetic.

To encrypt the plaintext into ciphertext, each letter in the plaintext is replaced by a letter that is some fixed number (the key) of positions down the alphabet. For example a right-shift of 4 would replace A with E and a left-shift of 3 would replace D with A. If the shift takes you past the end of the alphabet, you can loop back to the beginning. For example, a right-shift of 3 on Y would return B. We can therefore generalise this cipher as a function $f(x) = x + k(modn)$, where x is some number value designated to the letter (for 'b' this would be 2 for instance), k is the key, and n is the length of the alphabet.

To do this practically, this could be done by alligning two alphabets. The cipher alphabet is the plain alphabet rotated left or right by a number of positions. For example, here is a Caesar Cipher that uses a rotation to the left of three places (equivalent to a rotation to the right of 23):



Plain	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Cipher	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W

However, we can also implement this mathematically as suggested above. This can be seen implemented in the function here:

```python
def caesar_shift(plaintext: str, keyset: str, key: int) -> str:
    ciphertext = ""
    number_of_keys = len(keyset)

    for char in plaintext:
        if char.lower() in keyset:
            char = char.lower()
            index = keyset.index(char)
            new_index = (index + key) % number_of_keys
            char = keyset[new_index]
            ciphertext += char
        else:
            ciphertext += char
    return ciphertext
```

The Caesar Cipher was a useful way to protect the transfer of information thousands of years ago, when it was relatively new. However, because of its simplicity, people quickly realized that it was very easy to decrypt. This could be done by a brute force method of simply trying all possible keys (adding or subtracting all the numbers from 1 to 25) and seeing which one produced a readable message because the alphabet is so small. As shown computationally here, but using a different approach to give each letter numbers as it uses their ASCII value instead of putting them in a list):

```
def break_caeser(cypher, keys):
    newplain = ""
    for i in range (0, len(cypher)):
        if ord(cypher[i]) == 32:
            newplain = newplain + chr(32)
        elif ord(cypher[i]) < 97 or ord(cypher[i]) > 122:
            pass
        else:
            num = ord(cypher[i])
            newnum = ((num + keys - 97) % 26) + 97
            newplain = newplain + chr(newnum)
        return newplain
```

This could also be done by trying to deduce the key using frequency analysis, where common letters and words are compared to the cipher text to see what shift would be required to produce them and then this key is tested. The most common letter in the English alphabet is 'e', followed by 't', and 'a' and so on. Therefore, we could determine which letter is shifted to 'e' for instance simply by searching for the most common letter, allowing us to find the key for the caesar cipher.
Therefore, the Caesar Cipher no longer has any use in communication security. However, the encryption step using a key was developed into more complex ciphers such as the Vigenere Cipher.

1.2 Vigenere Cipher

The Vigenere Cipher is a more complicated, *polyalphabetic* substitution cipher where each letter of the plaintext is encoded with a different Caesar Cipher, determined by the corresponding letter of another word, the key. it is not simply monoalphabetic as the same letter in two different positions may be ciphered to different letters. the plaintext is split into sections each the length of the key. The first letter of the section will be encoded by a caesar Cipher shift of the numerical value in the alphabet of the first letter in the key. This continues in each section. For example, if the first letter of the plaintext in A and the first letter of the key is E then A will undergo a left-shift of 5 (as E is the 5th letter in the alphabet).

To do this computationally the plaintext should be split into sections (each the same length as the key length). This can be done by keeping a counter that increments each time a letter is Caesar shifted and if it gets larger than the length of the keyword will go back to 0. This counter can be used as the index to use the correct letter of the keyword to use for the Caesar shift. This can be seen being implemented here:

```
def vigenere_cipher(plaintext: str, keyword: str) -> str:
    ciphertext = ""
    current_cipher_index = 0
    length_of_keyword = len(keyword)

    for char in plaintext:
        if char.lower() in alphabet:
            if current_cipher_index > length_of_keyword - 1:
                current_cipher_index = 0
            local_key = alphabet.index(keyword[current_cipher_index].lower())
            ciphertext += caesar_shift(char, alphabet, local_key)
            current_cipher_index += 1
        else:
            ciphertext += char
    return ciphertext
```

This method of encryption is much more secure than the Caesar Cipher as each letter could be shifted by any of the 26 letters so without the key it is very hard to work out how to decrypt each letter. However, due to the repetitive nature of the key it can sometimes be possible to break this cipher.

One way this can be done by looking for repeated sections in the ciphertext and seeing how far apart they are from each other. The key length can then be assumed to be a factor of this distance. For example in the following ciphertext:

$$\boldsymbol{CSASTP}KVSIQUTGQU\boldsymbol{CSASTP}IUAQJB$$

The substring "CSASTP" is 16 letters apart. Therefore, it can be assumed that these repeated segments represent the same plaintext segments and so the key length is a factor of 16 (16,8,4,2,1). This is easier to implement with longer plaintexts as there are likely to be more repeated segments. The key length can then be guessed by looking at common factors of the distances between all the repeated segments. This method to guess the key length is known as 'Kasiski analysis' [1] which we can see implemented here:

```
def kasiski_analyser(ciphertext: str) -> list[int]:
    string_to_analyse = ""
    for char in ciphertext:
        if char != " ":
            string_to_analyse += char # remove spaces

    repeated_substrings = {} #substring: [positions]

    for substring_length in range(3, 12):
        for i in range(len(string_to_analyse) - substring_length - 1):
            potential_repeat = string_to_analyse[i:i+substring_length]

            positions = []
            index = 0
            while index < len(string_to_analyse):
                index = string_to_analyse.find(potential_repeat, index)
                if index == -1:
                    break
                positions.append(index)
                index += substring_length

            if len(positions) > 1 and potential_repeat not in repeated_substrings:
                if positions[0] % substring_length == positions[1] % substring_length:
                    repeated_substrings[potential_repeat] = positions

    distances = []
    for sub in repeated_substrings:
        distances.append(repeated_substrings[sub][1] - repeated_substrings[sub][0])

    d_factors = []
    for distance in distances:
        d_factors += factors(distance)
        # factors(n) returns a set of the factors of n

    # remove 1
    d_factors = [num for num in d_factors if num != 1]
    # sort into frequency
    d_factors = [i for items, c in Counter(d_factors).most_common() for i in [items] * c]
    # remove duplicates
    d_factors = list(dict.fromkeys(d_factors))
    # adds back one if the list is empty
    d_factors = [1] if d_factors == [] else d_factors

    return distances_factors
```

However, kasiski analysis has a higher liklihood to fail if the ciphertext is smaller, as it depends on repeating substrings, which are less likely to be present in short ciphertexts.

Once the length of the key is guessed, the ciphertext can be split into many columns, with each column corresponding to one letter of the key. Frequency analysis can then be used on each column where the most common letters in that column are compared to the most common letters in the english language to try and find the value of the Caesar shift in that column.

Overall, the Caesar Cipher is a very primitive cipher that is very easy to crack. The Vigenere cipher is more advanced but for longer plaintexts can almost always be solved by using frequency analysis. Therefore, neither of these methods are used in modern day communication.

1.3 AES and Symmetrical Encryption

...

1.4 RSA and Public-Key Encryption

RSA is an assymetric modern cipher that is currently in use, which uses the difficulty of prime factorisation to efficiently encrypt text. A *symmetric* cipher can be seen as follows, in order to demonstrate the flaws of this paradigm. We will call the sender of a message Alice and the receiver Bob. All classical ciphers rely on a key that both Alice and Bob know, which is in practice stored locally. This has some major disadvantages:

1. Alice and Bob need to meet to decide on a key
2. If someone else finds the key, they will be able to:
    - Read all encrypted messages
    - Send messages pretending to be Alice or Bob

Where an operation is described as *'easy'* it has time and space complexities with less than exponential growth.

Current Cryptography relies on a different approach, instead of Alice and Bob using the key, Alice locks her ciphertext with Bob's lock. So no-one (not even Alice) can unlock the message except Bob. This needs a lock that is extremely hard to break (to unlock without the private key) but very easy to unlock for Bob (the intended recipient*), as he of course has the key to his own lock.

We will call this "lock" a *public key* and the key that is secret a *private key.* A message can be encrypted by anyone using the public key and can only be decrypted by the person with the private key. The algorithm for generating the public and private keys should make it easy to:

- Generate public and private keys
- Decipher the ciphertext with the private key

But difficult to decipher the ciphertext without the private key This is called a *trapdoor algorithm.*

1.4.1 Key Generation, Encryption and Decryption

We assume that there is no easy algorithm for factoring large primes.

- To encrypt a message: $C \equiv M^e \mod n$
- To decrypt a message: $M \equiv C^d \mod n$

Where the following should be explained before the formula: C is the ciphertext (the encrypted message) M is the message (as an number)

We generate numbers n, e, and d such that:

- The values are computable (so the sender can encrypt their message)
- It must be difficult to calculate d given e, n and C (as the latter will all be publicly accessible and d would allow eavesdroppers to read the message)
- The message can be decrypted only by knowing d (so an eavesdropper can not read the message)
- $d =$ the private key
- $e =$ part of the public key
- $n =$ part of the public key
- n is chosen to be the product of 2 primes: p and q
- The difference of p and q should be high so that checking numbers near $\sqrt{n}$ (which is easy to calculate) does not reveal p and q

$$n = pq$$

We define $\phi(n)$ as the 'Euler Totient Function' of n, which is the number of numbers less than n which are coprime to n. Since n has a known factorisation (to the users of the key, but unknown to the third parties), we are able to efficiently calculate this as $(p-1)(q-1)$.

Then select d to be a coprime to $\phi(n)$.

Then select e such that:

$$ed \equiv 1 \mod \phi(n)$$

This ensures that e and d are multiplicative inverses of eachother $\mod n$, allowing us to use them as our private and public keys.

For example, let p and $q = 5$ and 31 respectively. Then $n = pq = 5 * 31 = 155$. In reality these numbers would be very large, though for the sake of demonstration we have ignored this. We can then choose d to be 7, as 7 is coprime to $(5-1)(31-1)$, or 120. (It happens to be coprime to every number, as it is prime itself). Then we can choose e as 103, as $103 * 7 \equiv 1 \mod \phi(n)$.
Using 7 and 103 as our private and public keys, we can encrypt, for example, the letter 'a', with ASCII value 97, as $97^7 \mod 155$, equal to 78, and decrypt this as $78^{103} \mod 155$, which is equal to 97.

1.4.2 The Hardness of Prime Factorisation

...

1.5 Shor's Algorithm

1.5.1 Quantum Computers

1.5.2 Prime Factorisation

2 Methods

Due to the security concerns of modern cryptosystems posed by quantum computers, there have been numerous attempts to develop new systems that are resistant to quantum algorithms, yet are still able to run on classical computers. Of these, NIST have identified four algorithms to be standardised.

2.1 Principles for Quantum-Resistant Cipher Development

Ciphers currently in use are based around prime factorisation, discrete logarithms and elliptic curves, all of which have an efficient quantum algorithm due to Peter Shor. Therefore, cryptographers have used different mathematical principles for developing future ciphers.

2.1.1 Worst-Case Lattice Problems

2.1.2 Learning With Errors

Suppose we are to recover a secret $\boldsymbol{s} \in \mathbb{Z}_q^n$, meaning s is an n-length vector of integers mod q. We are given that, for example:

$$5s_1 + 3s_2 + 10s_3 = 6 \mod 17$$

$$12s_1 + 9s_2 + 4s_3 = 14 \mod 17$$

$$13s_1 + 3s_2 + 8s_3 = 5 \mod 17$$

For classical computers, this is trivially easy to solve using Gaussian elimination, where all calculations are performed modulo 17. We can represent this system as a matrix as follows:

$$\left(\begin{array}{ccc|c} 5 & 3 & 10 & 6 \\ 12 & 9 & 4 & 14 \\ 13 & 3 & 8 & 5 \end{array}\right)$$

This is then reducible to the following matrix:

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 16 \\ 0 & 0 & 0 & 12 \end{array}\right)$$

This gives us the solution $s = \begin{pmatrix} 5 \\ 16 \\ 12 \end{pmatrix}$.

In fact, the time complexity of Gaussian elimination is $O(n^3)$, where n is the number of equations and variables, meaning simultaneous equations can be solved efficiently *polynomial time*, which is relatively quickly compared to other algorithms.

However, we can make solving simultaneous equations significantly harder to solve by adding some *error*. Suppose instead we are to recover secret $\boldsymbol{s} \in \mathbb{Z}_q^n$ and are given:

$$5s_1 + 3s_2 + 10s_3 \approx 6 \mod 17$$

$$12s_1 + 9s_2 + 4s_3 \approx 14 \mod 17$$

$$13s_1 + 3s_2 + 8s_3 \approx 5 \mod 17$$

$$\vdots$$

where each equation is correct up to some small additive error, say ± 2. If we try to use Gaussian elimination on a system of equations with error, then by performing operations on combinations of these equations we begin to amplify the error on our equations.

LWE is used as the basis for many PQR ciphers as there are strong reasons to believe that it is secure. One reason is that the best known algorithm has time complexity $2^{O(n)}$, even making use of quantum computers.
Another reason is that it is an extension of the *learning parity with noise* problem. The LPN problem is a special case of the LWE problem where equations are done modulo 2, so that the error can only be 1, occuring with probability ϵ. This is already an widely researched problem in learning theory that is widely believed to be hard, and allowing the modulus to be even greater than 2 increases the hardness. Furthermore, any algorithmic progress with LPN would be a breakthrough in coding theory, as it can be formulated as the problem of decoding from random linear binary codes.
The most significant reason is due to the ties LWE has with lattice problems. For example, we can assume that G_{AP}SVP, the decision version of the shortest vectors problem, is hard to approximate even with a 'hint' of a short basis, or we can assume that G_{AP}SVP or SIVP, the shortest independent vectors problem, is hard to approximate within polynomial factors even using quantum computers. Both of these assumptions we have enough reasonable evidence to believe.
For these reasons, we can trust the security of cryptosystems that use the principles of LWE against both classical and quantum algorithms.

2.2 Auxilliary Efficiency Algorithms

2.2.1 Fast Fourier Transform

2.2.2 Extended Euclidean Algorithm

3 NIST-approved Quantum-Resistant Ciphers

3.1 CRYSTALS Kyber

Kyber is a post-quantum cryptosystem which uses the LWE problem to safely establish a shared secret between two parties. It revolves around using lattices of polynomials to encrypt and decrypt messages.

3.1.1 Key Generation

The key generation algorithm has a series of parameters:

- n, the (maximum) degree of the polynomials;
- k, the order of the lattices;
- q, the modulus, which must be prime;

We can now define s and e as a two $k \times 1$ order matrices of polynomials with degree n and small coefficients, chosen through a binomial distribution. Then we define A as a $k \times k$ order matrix with elements of polynomials with degree n and random coefficients which are lesser than q. Finally, we define $t = As + e$, and can output (A, t) as our public key and keep s safe as our secret key. In the calculation of t, all operations are performed modulo q, and then modulo $x^n + 1$, to ensure that all coefficients remain lesser than q, and that the order of the resulting polynomials have degree lesser than n. The addition error polynomial e ensures that we cannot easily find s simply from A and t. We may, for example, define A, s, and e as follows, using parameters $n = 4$, $k = 2$, and $q = 91$:

$$A = \begin{pmatrix} 59x^4 + 64x^3 + 7x^2 + 16x + 46 & 49x^4 + 51x^3 + 73x^2 + 46x + 22 \\ 66x^4 + 65x^3 + 36x^2 + 34x + 54 & 28x^4 + x^3 + 76x^2 + 65x + 87 \end{pmatrix}$$

$$s = \begin{pmatrix} 2x^4 + x^3 + 2x - 1 \\ -x^3 - x^2 + 1 \end{pmatrix}$$

$$e = \begin{pmatrix} -x^4 + 2x^2 + x + 2 \\ -x^4 - x^2 \end{pmatrix}$$

It then follows that under modulo 91 and modulo $x^4 + 1$:

$$t = \begin{pmatrix} 24x^3 + 46x^2 + 79x + 26 \\ 54x^3 + 30x^2 + 69x + 12 \end{pmatrix}$$

We publicise (A, t) as our public key and keep secret s as our secret key.

3.1.2 Encryption

The encryption algorithms transforms a message, in the form of an integer, and a public key, which is a pair of matrices of polynomials, into a matrix of polynomials paired with a single polynomial as the ciphertext. The first step is to convert our integer message M into a polynomial M_b. We do this by taking the binary representation of the polynomial, then taking each single digit as a coefficient in a polynomial. For example, we convert the integer $13 = 8 + 4 + 1$ into the polynomial $(1)x^3 + (1)x^2 + (0)x^3 + 1$, or simply $x^3 + x^2 + 1$. Then, we multiply each coefficient by $\lfloor q/2 \rceil$, the nearest integer to $q/2$, to generate an upscaled message M_u. Then, we generate 2 random polynomial lattices , r and e_1, and a random polynomial e_2, all with small coefficients. From these values we can calculate our ciphertext like so:

$$u = A^T r + e_1$$

$$v = t^T r + e_2 + M_u$$

and output our ciphertext as the pair (u, v). The definition of u shows that it is not affected by the message M, though it is still sent in the ciphertext so that v can be decoded.
For example, we can randomly generate r, e_1 and e_2 as follows:

$$r = \begin{pmatrix} 2x^3 + 2x + 2 \\ -x^4 - 2x^3 + -x + 1 \end{pmatrix}$$

$$e_1 = \begin{pmatrix} -2x^4 + x^3 - 2x^2 + x + 1 \\ -1x^3 - x^1 \end{pmatrix}$$

$$e_2 = x^4 - 2x^3 - 2x^2 - 2x + 1$$

Using our example of encrypting $M = 13$, and continuing with our keys from the example in the previous section, we first find M_b as $x^3 + x^2 + 1$ as seen above. We then calculate M_u under modulo 91 and modulo $x^4 + 1$ by multiplying each coefficient by $\lfloor 91/2 \rceil = 41$, to get $M_u = 41x^3 + 41x^2 + 41$. Then we can calculate u and v like so:

$$\begin{aligned} u &= A^T r + e_1 \\ &= \begin{pmatrix} 53x^3 + 84x^2 + 54x + 17 \\ x^3 + 43x^2 + 23x + 1 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} v &= t^T r + e_2 + M_u \\ &= 17x^3 + 72x^2 + 29x + 17 \end{aligned}$$

3.1.3 Decryption

From our ciphertext (u, v), we first calculate a *noisy* message $M_n = v - s^T u$, which by substituting the definitions, we find that:

$$\begin{aligned} M_n &= v - s^T u && (1) \\ &= t^T r + e_2 + M_u - s^T(A^T r + e_1) && (2) \\ &= t^T r + e_2 + M_u - s^T A^T r + s^T e_1 && (3) \\ &= r(t^T - s^T A^T) + e_2 + M_u + s^T e_1 && (4) \\ & && (5) \end{aligned}$$

Then given that $t = As + e$, we have that $M_n = e^T r + e_2 + M_u + s^T e_1$, so we must still remove the error to retrieve the plaintext. This had been done in the encryption phase by upscaling the message, so the coefficientts in the message are large when compared to the *small* coefficients of the error polynomials. Thus we simply downscale the noisy message by first, for each of its coefficients, checking whether it is closer to 0 or $\lfloor q/2 \rceil$ or q and replacing it with this value (if it is closer to q then replace it with 0 because all calculations are performed modulo q), and then by multiplying each coefficient by $1/\lfloor q/2 \rceil$, leaving us with our binary message M_b. Finall to retrieve our plaintext we take the coefficients of M_b and convert it to denary.

For example, using our values of u, v and s in the previous sections, we can deduce the plaintext like so: First, calculating the noisy message:

$$\begin{aligned} M_n &= v - s^T u \\ &= 59x^3 + 42x^2 + 85x + 65 \end{aligned}$$

Then, downscaling the noisy message:

$$M_n \approx 41x^3 + 41x^2 + 0x + 41$$

$$M_b = 1x^3 + 1x^2 + 0x + 1$$

Then, finally converting this binary form into denary:

$$M = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 13$$

This successfully returns the plaintext.

3.2 CRYSTALS Dilithium

3.3 Falcon

3.4 SPHINCS$^+$

3.5 Comparison and Usability Criteria

4 Discussion

4.1 Comparison of Quantum-Resistant Ciphers

4.2 Overall Usability of Quantum-Resistant Ciphers

5 Conclusion

...

6 Programme of Work

6.1 Meeting 1 (16.10.24)

In our first meeting we were introduced to classical cryptography with the Caesar and Vigenere ciphers. We were also given a small introduction to RSA encryption. We were tasked with programming and solving the Caesar and Vigenere ciphers and investigating RSA encryption in python. The Vigenere cipher was much more difficult to decode than the Caesar cipher, as the size of the Caesar cipher's keyset is 26 while the size of the keyset of the the Vigenere cipher is 26^n where n is the length of the ciphertext. We first tried to decode it using the Index of Coincidence [2] of columns of the ciphertext, which can be used to guess the length of the keyword, after which frequency analysis can be applied. This method was unsuccessful, and we then opted to Kasiski analysis [1], which was much more succesful. Following this, we were able to decode the Vigenere cipher to a large degree of accuracy - on certain ciphertexts, particular of a small length, have a lower chance of being correctly decoded due to the nature of the Vigenere cipher, being easier to decode with larger texts.

6.2 Meeting 2 (6.11.24)

During our second meeting, Mr Savage reviewed our code and guided us on making it more pythonic and instructed us to add docstrings. We also discussed the specific details of our long-term goal of our project, and we decided to target quantum computers, as we believe that these pose the greatest threat to quantum cryptography. After the meeting, we updated our code according to Mr Savage's guidelines, and proceeded to research quantum algorithms. RSA encryption's security depends on the hardness of integer factorisation, (which on classical computers has time complexity big O, while quantum computers have time complexity big O), due to Shor's algorithm, which can theoretically be used to factorise integers very quickly once there are powerful enough quantum computers (insert value of how powerful). We also began to research 'learning with errors', which is the basis of many quantum-resistant algorithms [3].

6.3 Meeting 3 (20.11.24)

During our third meeting, Mr Savage provided us with a Github repository of a lattice-based learning with errors cipher, with guidance on what research we could pursue.

References

[1] Kasiski's method. `https://pages.mtu.edu/~shene/NSF-4/Tutorial/VIG/Vig-Kasiski.html`. Accessed: 2024-12-03.

[2] Index of coincidence. `https://pages.mtu.edu/~shene/NSF-4/Tutorial/VIG/Vig-IOC.html`. Accessed: 2024-12-03.

[3] O. Regev and D. Micciano. Lattice-based cryptography. `https://cseweb.ucsd.edu/~daniele/papers/PostQuantum.pdf`. Accessed: 2024-12-03.