BENCHMARKING PRE/POST-QUANTUM CRYPTOGRAPHY

BY

Cavan Phelan

C00249198

17 April 2023

Contents

1. Introduction

Cryptography has been around longer than we could imagine. It is believed to be used as far back as 100 BC (Sidhpurwala, 2023). Julius Caesar used the Caesar Cipher to communicate with his commanders in war. This cipher used the alphabet and shifted the characters by an amount only his commanders knew. Cryptographic algorithms have changed in complexity and methods since then with the advancement of technology. Despite this, the main goal of cryptography then and now is somewhat the same which is to communicate and exchange information securely without the interference of third parties.

We are at the stage where we are about to enter a new age of cryptography, with the main cause being quantum computers. Quantum computing is highly anticipated to break many encryption algorithms such as AES, RSA and 3DES. Any algorithm that relies on computer mathematical problems will such as the integer factorization problem, discrete logarithms problem and elliptic curve discrete logarithm problem are quantum unsafe, all of which will be explained later. Post-Quantum algorithms also rely on some computational problems, but these cannot be solved efficiently on classic and quantum computers.

2. Benchmarking Goals

There is a selection of categories I wish to benchmark to get a good representation of whether post-quantum algorithms are ready for use now, and if so, determine if they are more reliable and dependant than classic algorithms.

Performance – Benchmarking the amount of time taken to perform certain tasks such as encryption, decryption, signing and verifying signatures. A benchmark of the average time of these after a certain number of runs would help me determine a good baseline.

Security – Benchmark and analyse the output of the algorithms, whether it's with key sizes, plaintext and ciphertext lengths and signature lengths. Getting a good idea of the size of these would help determine the storage needs of each.

Parameters – The number of customizable parameters to change the level of security and performance of the algorithms. These benchmarks would help see the impact of different key sizes and how they affect each algorithm.

These main parameters would help build a baseline of what is considered fast and compact in terms of both classical and quantum-resistant algorithms and see if the increase in computation need, and storage is worth the trade-off.

3. Classic Algorithm Security

Algorithms such as AES, RSA and Diffie-Hellman all rely on three mathematical problems to maintain security. These problems are (3.1.1) **integer factorization problem,** (3.1.2) **discrete logarithm problem and** (3.1.3) **elliptic-curve discrete logarithm problem.**

3.1. Integer factorization

This is determining which prime numbers divide a given positive integer. The bigger the size of the number, the more resources are needed to factorize it (OBE, 2018). It is currently very hard and expensive for a computer to factorize large semiprimes (Product of only two prime numbers), or any integer that has no small factors.
To put this into perspective, in 2020, RSA-250 which has 250 decimal digits (829 bits) was factored. Even with today's technology, this took around 2700 CPU core years to do. RSA-260 has not been factored in, showing that even a small increase in decimal digits means a lot more computer resources (Schneier, 2020).

3.2. Discrete logarithm

This uses modular arithmetic as it is a numerical procedure that is easy in one direction but hard in the other. If we take a prime number p, and we take x to be a primitive root of p, we can use x mod p to return a value. For example, if we use 3^1 mod 17, we get 3. 3^{16} mod 17 = 1. If we take the prime number (p-1) and take one away from it, that is how many results we can produce. If we mod the value 3 by 17, then we can have results from 1-16. If we used a prime of 77, we could get a value from 1-76. All these numbers, from 1-76 all have an equal chance of being the correct answer, making it hard to just guess which might be correct.
This is easy to solve on small numbers but if we receive 3^n mod 7757 = 3552, how do we calculate n? We would have to brute force the value of n from 1-7756 until we find the correct value of n, so the equation equals our result. This is expensive to do on a computer. As mentioned above this took 2700-core years to brute force this on modern technology on a 250 decimal digit (Vacca, 2014).

3.3. Elliptic-curve discrete logarithm

This problem is considered very computationally hard to solve. A general algorithm for computing logb a in finite groups G is to raise b to larger and larger powers k until the desired a is found. ("Discrete logarithm - Wikipedia") This is also called triple multiplying, as you must keep adding and multiplying b to find the correct point, a, in the finite field. It requires running time linear (The more operations the harder it is to solve) in the size of the group G and thus exponential in the number of digits in the size of the group. It is an exponential-time algorithm (Running time grows as an exponential function of the size of its input). This becomes a problem when using more and larger numbers, according to the exponential-time algorithm mentioned before.

3.4. What puts these popular algorithms at risk?

Due to the above algorithms relying on the three mathematical problems, if these can be efficiently calculated then they are no longer secure. Shor's algorithm, which when run on a quantum computer can efficiently solve all these issues. The algorithm can find the prime factors of an integer. This has existed since 1994, but we lack the computational power to run it. This algorithm will be explained further in the document.

Types of classic algorithms

3.5. Hashing Functions

A hash function is used to map data of arbitrary to data of a fixed size. For example, if I hashed the word "hello" using SHA-256, the hash length will always be 256-bit. A hash is one-way and cannot be decrypted to the original state. Hashes must maintain two security guarantees to uphold its security, which is as follows.

a) **Collision resistant** – This is a security guarantee that it's hard to find two messages that hash to the same value. Collisions are however inevitable due to the Pigeonhole principle. The Pigeonhole principle states that if you have m holes, and n pigeons if n is greater than m, at-least one hole must contain multiple pigeons.

b) **Preimage Resistance** – This is the security guarantee that given a hash value, an attacker will never find a preimage (Original value) of that hash value. Preimage resistance breaks into two sections:

1. First-preimage resistance describes that given the hash value, it is practically impossible to find a message m that hashes to hash H.

2. Second-preimage resistance describes that given a message m_1, it's impossible to find another message m_2 that hashes to the same value m_1 does.#

3.6. Asymmetric-Key Algorithms

Asymmetric-key algorithms are also known as public-key systems. These systems generate a key pair. Each key pair consists of a public key and a private key. As suggested, the public key can be public and given to anyone who wishes to message you. However, only I can decrypt that encrypted message using my secret private key.

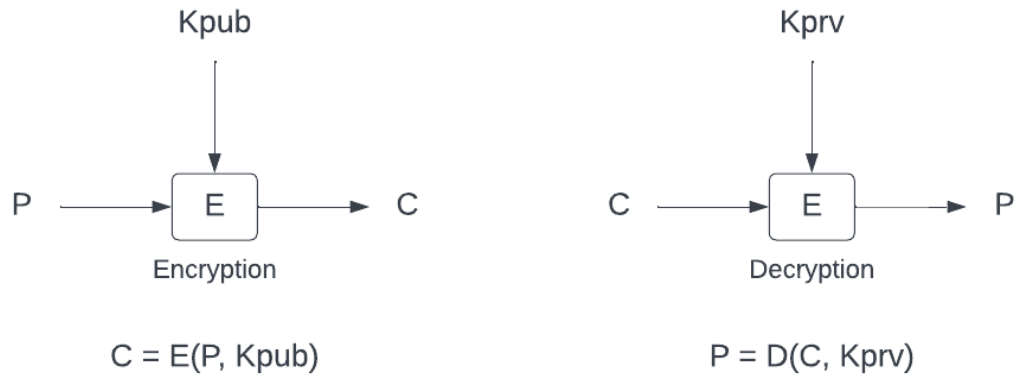


Figure 1: Basic Public-key algorithm

These Asymmetric-Key algorithms can usually also be used to create digital signatures. You can create a digital signature using your secret private key. Anyone can verify the signature using the sender's public key. Digital signatures provide authentication (the sender is

verified), integrity (the message hasn't been tampered with), and non-repudiation (the sender cannot deny sending the message).

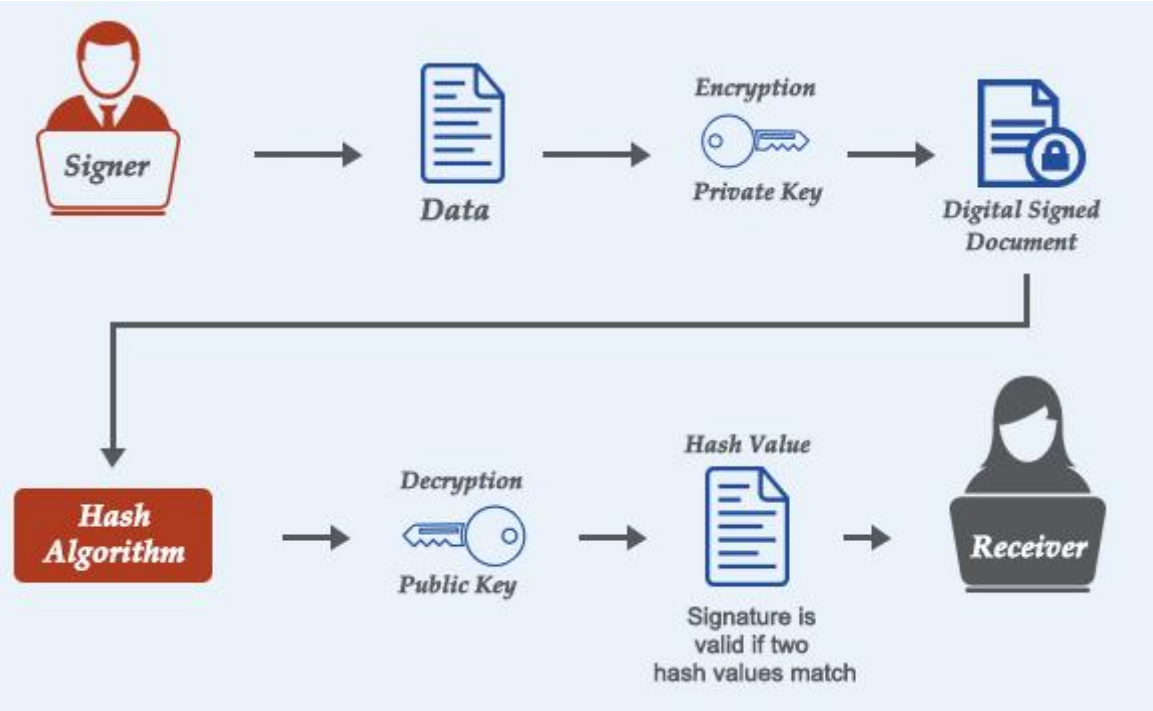


Figure 2: Digital Signature Example

3.6.1. How do you generate a key pair?

There are many different methods based on what algorithms you want to use and how you want to use them. Elliptic Curve Digital Signature Algorithm is one of the most efficient methods of generating a keypair and is used later in classical algorithms and even quantum.

We first choose an elliptic curve and a set of domain parameters that define the characteristics of the curve, such as the size of the key, the prime modulus, and the base point. Examples of commonly used elliptic curves include secp256k1 and secp384r1. We then choose a random number (the private key) that is within the range [1, n-1], where n is the order of the elliptic curve. After that, we multiply the base point of the elliptic curve by the private key using scalar multiplication to obtain the public key. The resulting point on the curve is the public key. Once done, we usually encode the key using the X.509 format.

3.6.2. What are Elliptic Curves?

Elliptic curves are a type of mathematical curve in the equation form y^2 = x^3 + ax + b.

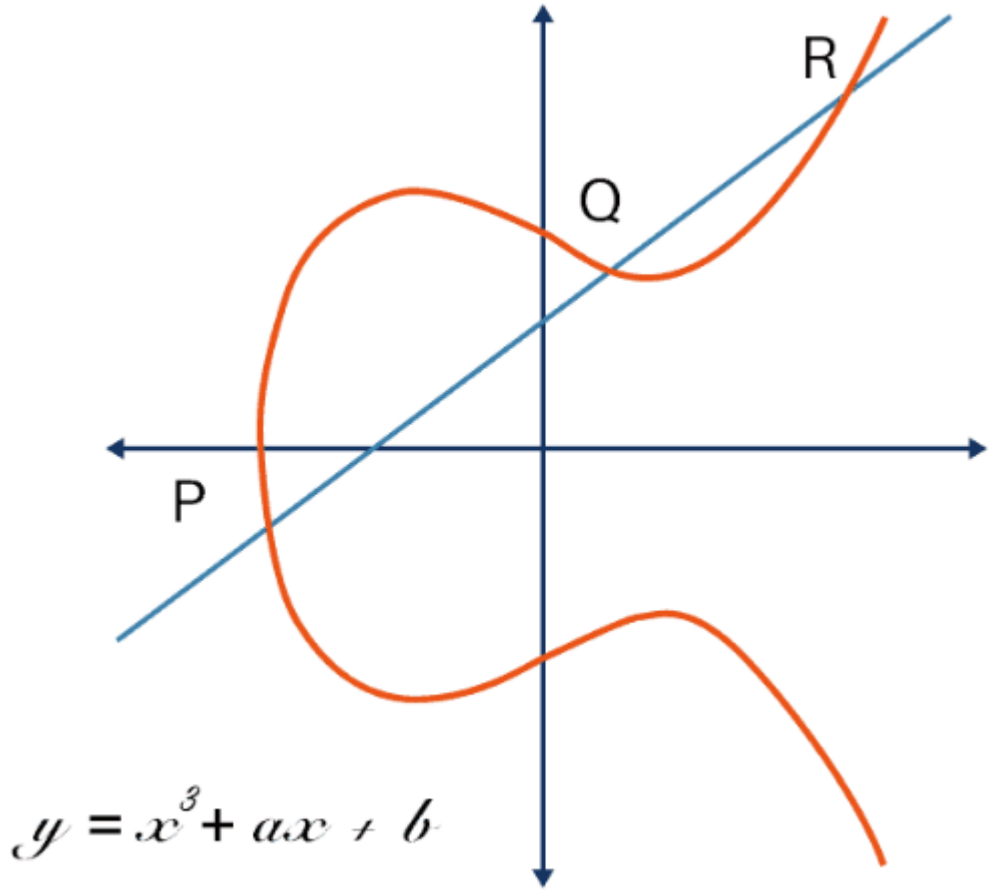


Figure 3: Elliptic Curve Example

where a and b are constants, and x and y are variables that satisfy the equation. The curve is defined over a finite field, which means that only a finite number of points on the curve have coordinated with values in the field. Elliptic curves have the property that it is computationally difficult to find the discrete logarithm of a point on the curve.

3.7. Symmetric-Key Algorithms

This is like Asymmetric-Key algorithms except we use the same key for both encryption and decryption. This means that encryption and decryption are very fast, but one of the main challenges of this is to securely communicate a common shared secret to communicate. If this secret key is exposed, communication is compromised. To combat this, we can use a Key-Exchange Scheme such as Elliptic Curve Diffie-Hellman Key Exchange (ECDH).

3.7.1. Elliptic Curve Diffie-Hellman Key Exchange

ECDH is a key agreement protocol that allows two parties to securely establish a shared secret key over an insecure channel without transmitting the key itself., it's based on the original Diffie-Hellman protocol. ECDH relies on the mathematical properties of modular exponentiation (an operation that involves raising a number to a power and taking the remainder of the result when divided by a specified modulus) to establish the shared secret key. ECDH uses points on an elliptic curve over a finite field.

The basic steps are as follows: 1. Each party generates their public-private key pair, consisting of a private key and a corresponding public key. 2. The parties exchange their public keys over the insecure channel. 3. Each party uses their own ate key and the other party's public key to compute a shared secret key. This is done by multiplying the other party's public key by their privkey, which results in a shared point on the elliptic curve. 4. The parties can use the shared point to derive a shared secret key that can be used for symmetric-key encryption.

3.7.2. Diffie-Hellman Key Exchange

This works somewhat the same as the newer ECDH but uses integers rather than elliptic curves, is less efficient and has bigger key sizes.

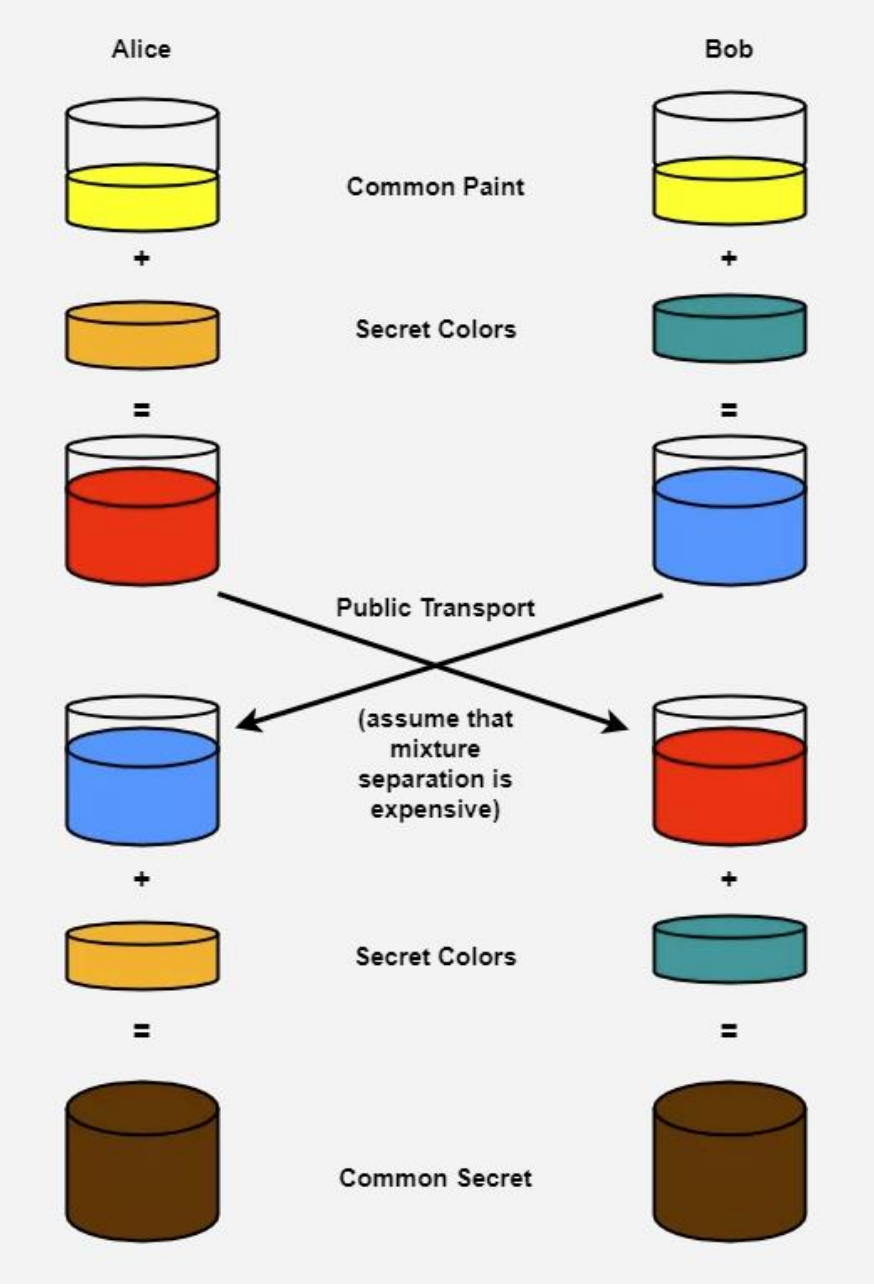


Figure 4: Diffie-Hellman Key Exchange

4. Classic Cryptography

4.1. Advanced Encryption Standard (AES)

Advanced Encryption Standard (AES), also known as **Rijndael**, is a variant of a block cipher. ("Advanced Encryption Standard (AES) in Counter Mode") Block ciphers take in a data 'block' and processes them. These blocks are 128 bits in length. We can vary our key sizes, which changes how many rounds AES goes through:

- 10 rounds for 128-bit keys.
- 12 rounds for 192-bit keys.
- 14 rounds for 256-bit keys.

AES is a symmetric-key algorithm, meaning that the same key is used for encrypting and decrypting data. AES is based on a design principle called the 'Substitution-permutation network (SPN)'. This is a series of mathematical operations used in block ciphers. The plaintext and key will be applied through several rounds of (a) **substitution boxes** (S-boxes) and (b) **permutation boxes** (P-boxes):

a) An **S-box** substitutes a small block of bits (input) with another block of bits (output). This substitution is one-to-one to allow for decryption. The length of the output should be the same as the input. An S-box should have the property that changing one bit will change about half the output (Avalanche effect).

S_5		Middle 4 bits of input															
		0000	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**	**1010**	**1011**	**1100**	**1101**	**1110**	**1111**
Outer bits	**00**	0010	1100	0100	0001	0111	1010	1011	0110	1000	0101	0011	1111	1101	0000	1110	1001
	01	1110	1011	0010	1100	0100	0111	1101	0001	0101	0000	1111	1010	0011	1001	1000	0110
	10	0100	0010	0001	1011	1010	1101	0111	1000	1111	1001	1100	0101	0110	0011	0000	1110
	11	1011	1000	1100	0111	0001	1110	0010	1101	0110	1111	0000	1001	1010	0100	0101	0011

Figure 5: S-box example

Given a 6-bit input, the 4-bit output is found by selecting the row using the outer two bits, and the column using the inner four bits. For example, an input '**0**1101**1**' has the outer bits '**01**' and the inner bits '1101'. The corresponding output would be '1001'.

b) A **P-box** is a permutation (Different combination) of all the bits. It takes the output of the S-boxes of one round, permutes the bits, and feeds them into S-boxes in the next round. ("The exact difference between a permutation and a substitution") P-boxes should have the property that the output bits of any S-box are distributed to as many S-box inputs as possible.

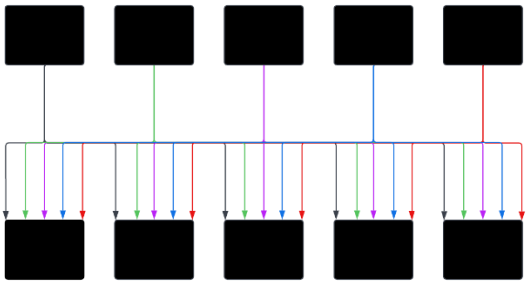


Figure 6: P-box example

At each round, the round key (Obtained by putting the key through S-boxes and P-boxes) is combined using some group operation, usually XOR, which is multiplying 0s and 1s together.

AES operates on a 4x4 column-major order array (Method of storing multidimensional arrays in storage like ram) of 16 bytes, from $b_0, b_1, \ldots, b_{15}$ which is known as the 'state'.

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

Figure 7: 4x4 column-major order array

Each round goes through multiple processing steps, including the encryption key also. This is done in reverse to decrypt the ciphertext. The processing steps are the following:

4.1.1. SubBytes

In this step, $a_{i,j}$ in the state array is replaced with a SubByte $S(a_{i,j})$ using an 8-bit substitution box. The initial round uses the input as the state array to provide non-linearity.

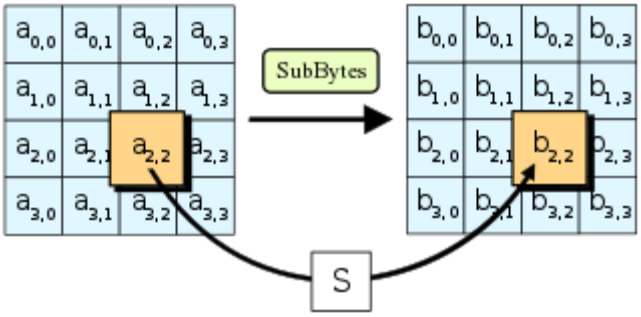


Figure 8: SubBytes step

4.1.2. ShiftRows

This step operates in the rows of the state. It shifts the rows by a certain offset. The first row is left unchanged. Each byte of the second row is shifted by one to the left. ("Advanced Encryption Standard - Wikipedia") The third row is shifted by two to the left and the last row is shifted by three. This step avoids the columns being encrypted independently.

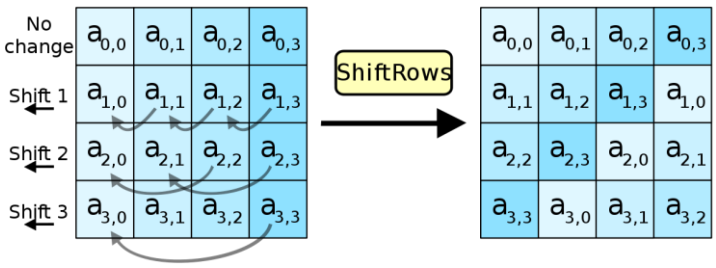
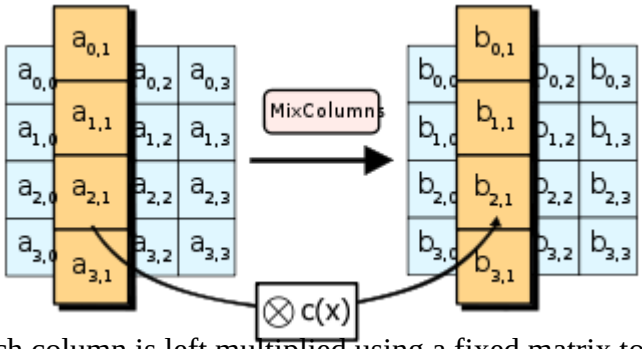


Figure 9: ShiftRows step

4.1.3. MixColumns

The four bytes from each column of the state are combined using a linear transformation. This function takes four bytes as input and outputs four bytes, where each input byes affects all four output bytes. Together with the previous two steps, this creates diffusion (Changing one bit, changes about half of the bits in the ciphertext) which hides statistical relationships between the ciphertext and plaintext.



During this operation, each column is left multiplied using a fixed matrix to give a new value of a column in the state.

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{bmatrix} \qquad 0 \leq j \leq 3$$

Figure 10: Columns being left multiplied using a fixed matrix.

4.1.4. AddRoundKey

The subkey is combined with the state. For each round, a subkey is created from the main key using a key schedule. Each subkey is the same size as the state. The subkey is added by combining each byte of the state with the corresponding byte of the subkey using XOR. (“Advanced Encryption Standard - Wikipedia”) (“Advanced Encryption Standard - Wikipedia”)

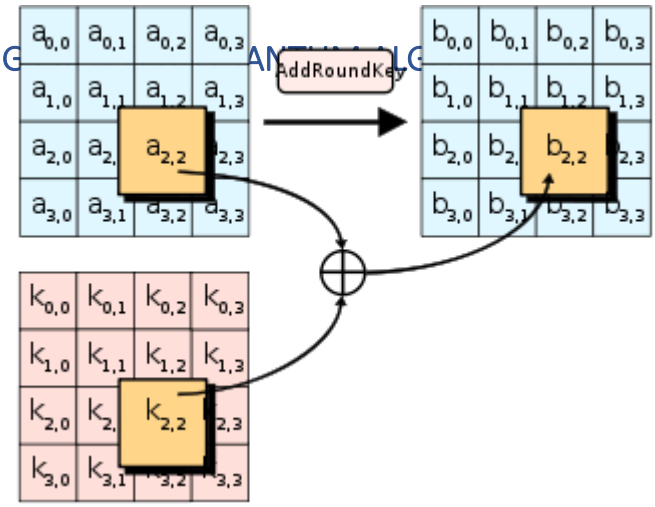


Figure 11: AddRoundKey example

Not all rounds include all the steps and are as follows.

a) The first round of AES is the *Key Expansion* round, where round keys are created by AES.
b) The second round only uses the *AddRoundKey* which produces the first bit of ciphertext to get started.
c) All the remaining rounds, except the very last use all the steps mentioned above.
d) In the last round, *MixColumns* is not included, as it does not make a difference.

4.1.5. Can we make AES more efficient?

It is possible to make AES faster and more efficient, and this is by choosing our mode of operation. We have two types of ciphers, (a) **block ciphers** and (b) **stream ciphers**.

a) A **block cipher** is a method of encrypting data in a block to produce ciphertext using a key and algorithm. ("Security Definitions from TechTarget") These block sizes are fixed size, usually 128 bits. (REF)

b) A **stream cipher** is a method of encrypting data bit by bit, which is faster than in blocks. (REF)

These modes of operation change the encryption process but do not change the rounds AES has. The most widely used mode of operation is the Counter (CTR) mode, which is the fastest mode. It turns a block cipher into a stream cipher by encrypting byte by byte rather than in blocks (Mustafeez, 2015).

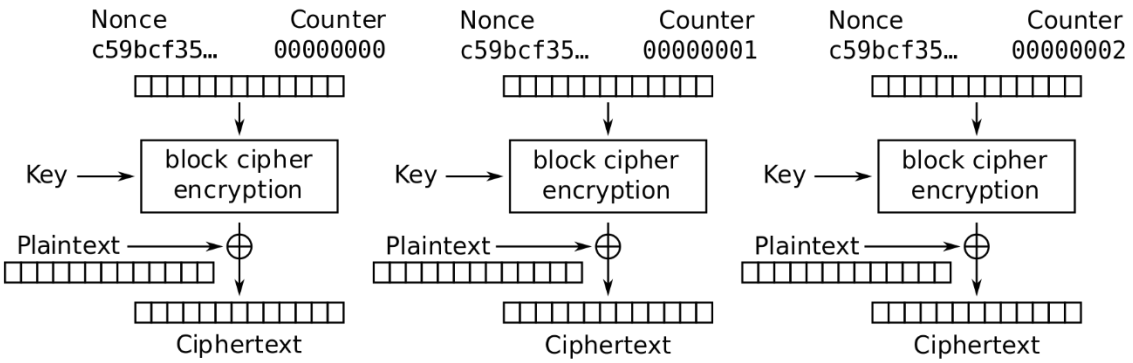


Figure 12: Counter Mode Encryption

To decrypt this, rather than XOR with the plaintext after the AES encryption, you XOR with the ciphertext to produce the plaintext.

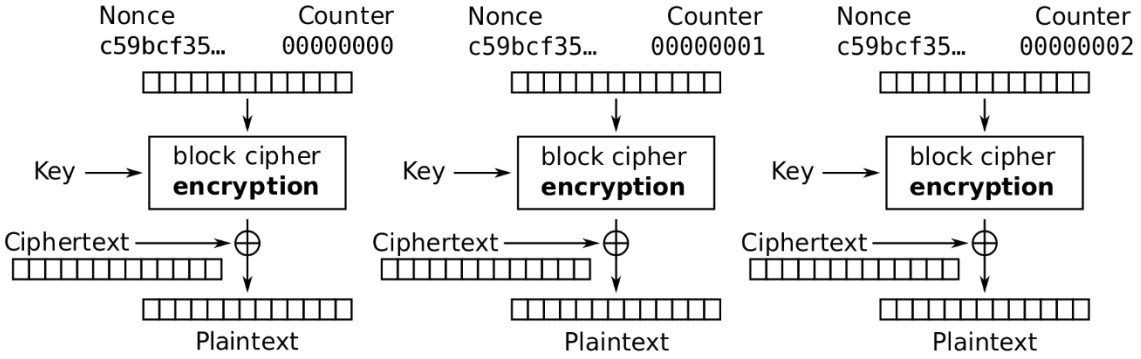


Figure 13: Counter mode Decryption

4.1.6. How does Counter Mode work?

Rather than using an initialisation vector (IV), which is a pseudorandom string of numbers, we can use a nonce (One-time, unpredictable value) and combine it with a counter to create a random input every time. This counter can be random, but it is popular to increment it if the nonce is random. The nonce and counter go through the block cipher encryption with the key and produce an output that is XOR'd with the plaintext to create the corresponding ciphertext.

4.2. Rivest–Shamir–Adleman (RSA)

RSA is a public-key cryptosystem. Public-Key cryptography uses a public key with a corresponding private key, otherwise known as a key pair. These keys are generated using mathematical problems called one-way functions, or trapdoors (Easy to compute but hard to reverse). In this system, anyone with a public key can encrypt a message, but only those who know the corresponding private key may decrypt the message. ("Public-key cryptography - Wikiwand")

4.2.1. RSA Key Pair Generation

1. I must choose two random large prime numbers, p, and q. These values must be kept secret.
2. Multiply your two prime numbers together to calculate n, n = pq. This is the modulus for the key pairs.
3. We use Carmichael's totient function, denoted as λ, to calculate $\lambda(n)$. The purpose is to find the smallest positive divisor of Euler's totient function. Euler's totient function counts the number of positive integers less than n but that are also coprime ton. The formula to calculate this is: $\lambda(n) = lcm(p-1)(q-1)$. LCM is the lowest common multiple, which is calculated from Carmichael's totient function. $\lambda(n)$ is kept secret.
4. Choose a new integer e. This integer meets the specific requirements as follows: $1 < e < \lambda(n)$. This means the number must be between 1 and λ(n). It also must be coprime with n and $\lambda(n)$. Integer e is released with the public key.
5. Determine d as de(mod(λn)) = 1. This is a secret as it's used in the private key.

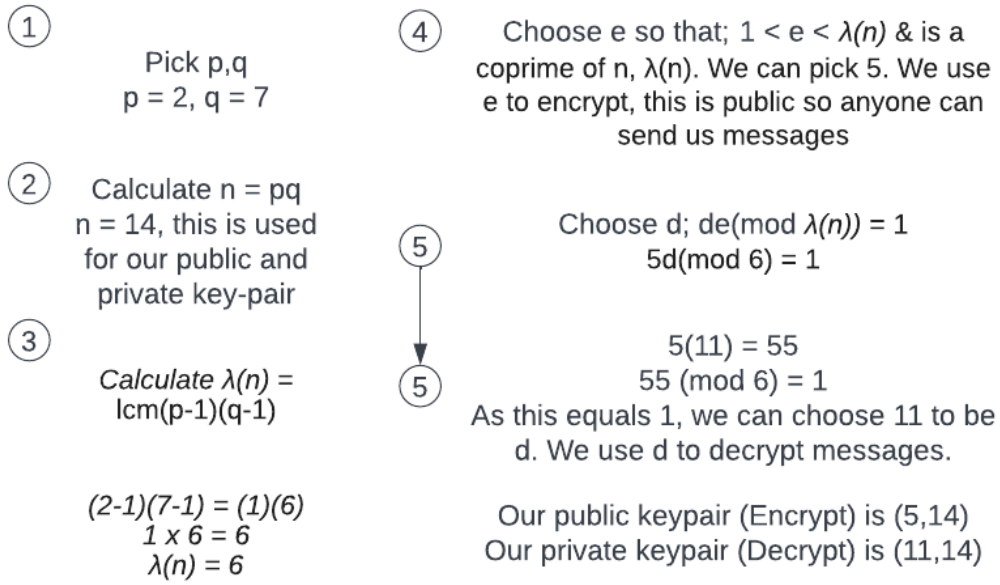


Figure 14: Key-pair generation example

Now that we have determined the values of our key pairs, we can begin to encrypt or decrypt messages (Java T Point, n.d.).

4.2.2. RSA Encryption

Encrypting a message is very quick and easier, especially compared to generating the key pair. For this example, I will use the alphabet as a guideline to convert characters to numbers. I want to send the message 'rot', which is converted into 12,5,4. I will reuse the keypair example from the previous section in the example (5, 14). To convert this to ciphertext I need to raise each number to the power of the first number in our keypair (5) and modulo by the second (14). L would be 12^5 (mod 14) = 10, E would be 5^5 (mod 14) = 3 and D would be 4^5 (mod 14) = 2. Our ciphertext would be 10,3,2, which converts back to JCB.

4.2.3. RSA Decryption

Decrypting a message is very similar to encrypting it. We will use the example from encrypting to decrypt. We will take our private key-pair (11, 14) and use it on the ciphertext, which is 10,3,2. We use the same formula as encrypting to decrypt. 10^{11} mod (14) = 12, 3^{11} (mod 14) = 5, 2^{11} (mod 14) = 4. If we convert 12,5,4 then we get an LED confirming we have successfully decrypted the message.

4.2.4. RSA Signatures

Let's say Alice wants to send an encrypted message to Bob. How can Bob be sure it is Alice? This information isn't included in the transportation of the message so Alice could be anyone, someone could pose to be Alice in the future, how do we prevent this?
With signatures, we can sign the message to prove our identity. Alice can produce a hash value of the message and raises it to the power of d, which is part of her private key as mentioned previously. She attached the signature to the message, the hash. When Bob received the message, he uses the same hash algorithm in conjunction with Alice's public key. He raises the signature to the power e, part of Alice's public key and compares the hash value to that of what he received. If the hash results match, Alice has verified her identity which means she had access to her private key. This also proves that the message has not been tampered with since being sent. Hashing with be explained further during a different algorithm (Geeksforgeeks, 2021).

4.3. Sha-256 (Secure Hashing Algorithm)

As mentioned earlier, hashing takes data of arbitrary size and maps it to data of a fixed size.

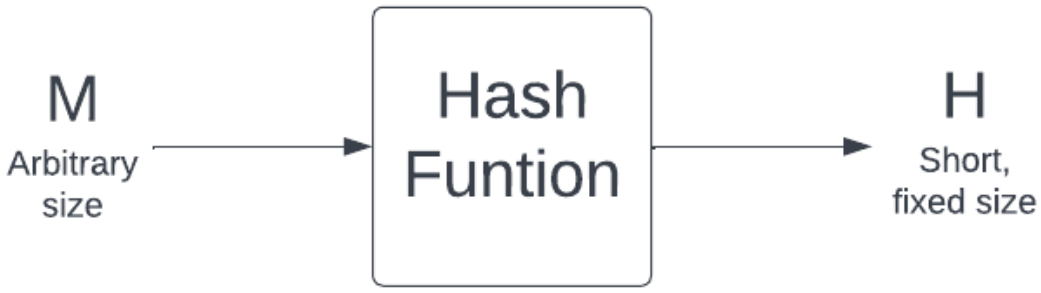


Figure 15: Basic hash algorithm

SHA-256 consists of seven steps to generate the final hash output and they are.

4.3.1. Pre-Processing

We first need to take our message and convert it into binary. For this example, I will convert ‘Cavan’. Each letter is 8-bits long.

01000011 01100001 01110110 01100001 01101110

Figure 16: 'Cavan' converted to binary.

Once this is done, the next step is to append a single ‘1’ to the end of the message.

01000011 01100001 01110110 01100001 01101110 1

Figure 17: A single '1' appended to the message.

We now pad the message with 0’s, until the message is a multiple of 512 (512 in this case). We still must make sure to leave the last 64 bits free for now so we would only pad 407 bits.

For the remaining 64 bits, we represent the length of our message here. Our message was 40 bits long, we store this as a big-endian integer (EXPLAIN), which is ‘101000’. With that done our message should now look like this:

```
01000011 01100001 01110110 01100001
01101110 10000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00101000
```

Figure 18: Message block after step 1.

The digits in green are our original message, the one bit in blue represents the single ‘1’ we append at the end of our message and the red is for the length in bits, of our message.

4.3.2. Initialize Hash Values (h)

We need to create eight hash values; we use the first eight prime numbers as a reference and assign them to a hash (H). What we need to do is get the root of these prime numbers. We then take the first 32 bits of the fractional part, multiply that by 2^{32} and convert from decimal to hex to get our result for the hash.

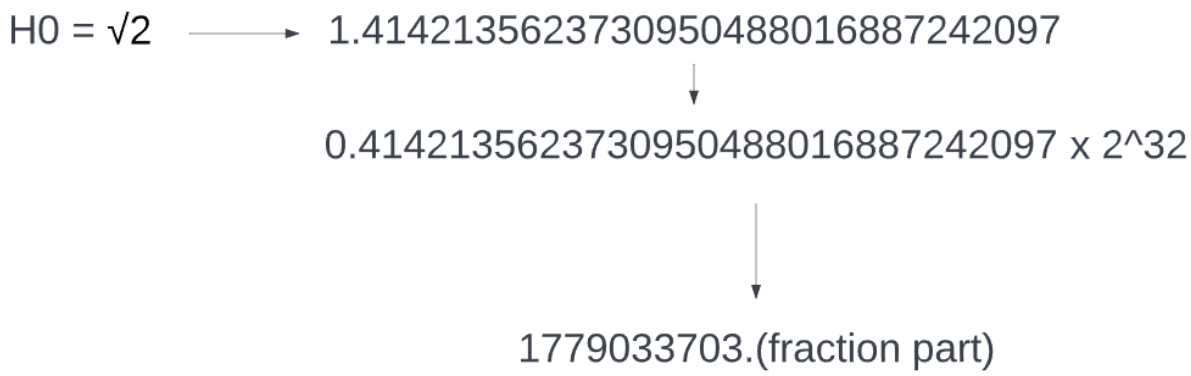


Figure 19: Calculating H0

Once we power the root of 2 by 2^{32}, we are no longer interested in the fraction part rather we want the whole number now. If we convert this decimal into hexadecimal, we will get 6a09E667, which is now the value of H^0. The result from the second prime number would be H^1 and so on.

4.3.3. Initialize Round Constants (k)

Creating round constants is the same process as above, except these are done with the first 64 prime numbers and assigned to k^0 up to k^{63}.

4.3.4. Create a Message Schedule

We take the message block we currently have and reorganise it in a way that each entry is now a 32-bit word.

```
w0  01000011011000010111011001100001
w1  01101110100000000000000000000000
w2  00000000000000000000000000000000
w3  00000000000000000000000000000000
w4  00000000000000000000000000000000
w5  00000000000000000000000000000000
w6  00000000000000000000000000000000
w7  00000000000000000000000000000000
w8  00000000000000000000000000000000
w9  00000000000000000000000000000000
w10 00000000000000000000000000000000
w11 00000000000000000000000000000000
w12 00000000000000000000000000000000
w13 00000000000000000000000000000000
w14 00000000000000000000000000000000
w15 00000000000000000000000001101000
```

Figure 20: Message Block in 32-bit words

W^0 up to W^{15} is filled with our original message block, but it doesn't end here. It goes all the way up to W^{63} but is not shown above as everything after W^{15} is only filled with 0s.

We now run a loop starting at W^{16} and ending at W^{63} and it does a sequence of calculations to start filling these empty blocks. We will do W^{16} as an example.

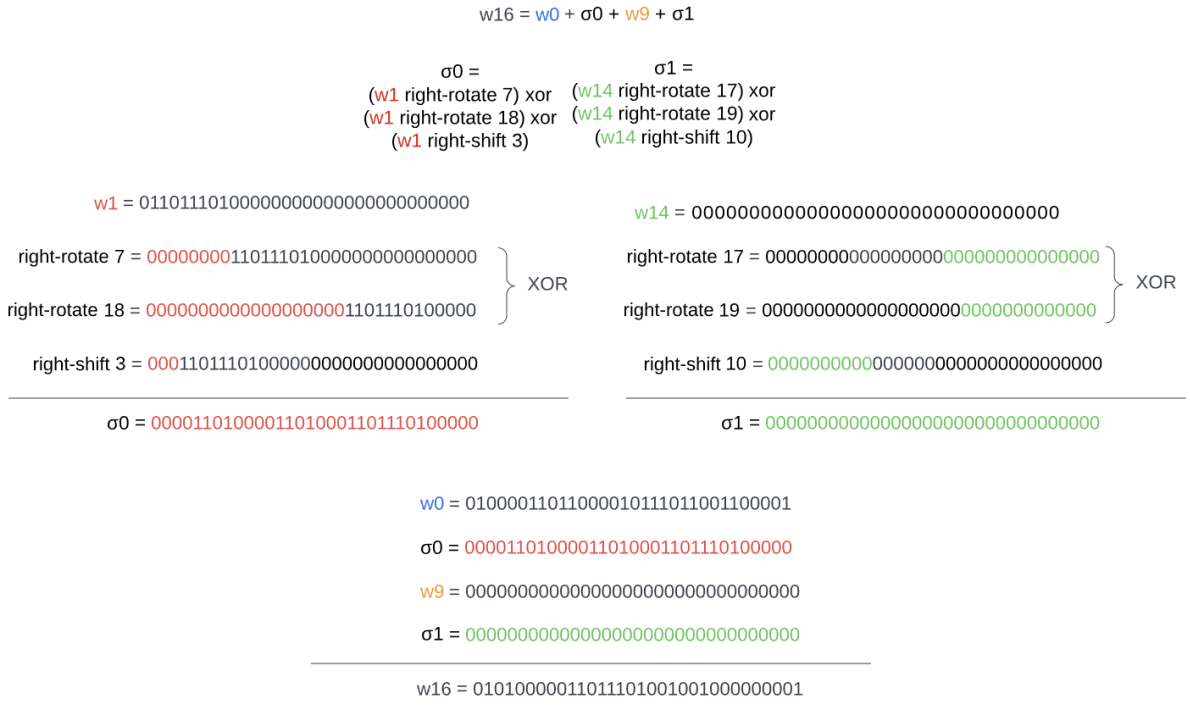


Figure 21: Example of how SHA-256 calculates W16.

Once complete, this moves to W17 and increments W0 -> W1, W9 -> W10, W1 -> W2, W14 -> W15.

Once complete this is what the message schedule looks like:

w0	01000011011000010111011001100001	w32	10101110000000010110000010001010
w1	01101110100000000000000000000000	w33	01100110101010010010100111001110
w2	00000000000000000000000000000000	w34	00101001010110100010010111001101
w3	00000000000000000000000000000000	w35	11111010101010001111111010010101
w4	00000000000000000000000000000000	w36	11110101001000000000110100111101
w5	00000000000000000000000000000000	w37	00001100101101001000110000110001
w6	00000000000000000000000000000000	w38	10110011010100010111100011101000
w7	00000000000000000000000000000000	w39	01010111110000001101000001010000
w8	00000000000000000000000000000000	w40	11011001011100101101111100010111
w9	00000000000000000000000000000000	w41	11100101101011110110110111100111
w10	00000000000000000000000000000000	w42	01000100110000100000010011011110
w11	00000000000000000000000000000000	w43	10000000111001000110111100110100
w12	00000000000000000000000000000000	w44	00011111101101111100000001101111
w13	00000000000000000000000000000000	w45	01101011011011111010001011011111
w14	00000000000000000000000000000000	w46	01110110000100101111011010110110
w15	00000000000000000000000000101000	w47	10110110110110100000111100001000
w16	01010000011011101001001000000001	w48	10010011101011010010111010100111
w17	01101110100100010000000000000000	w49	00000111101111101110010010000101
w18	10011011010101001001100110011110	w50	11100111111001111001011001101110
w19	10100000000110111001111011011010	w51	00101111001101011100100010001010
w20	11011111110110100100101111100110	w52	11011010110101101100001010111001
w21	10111100100111100000001011101001	w53	01111001010111011100001100101101
w22	01101100101110000100001010101100	w54	11000001101011000111100111001011
w23	00010001011101010110000001011101	w55	11001000101101001000101111101100
w24	10010111101010011001010111011011	w56	10011101001001100000000110100101
w25	10110111011101100001000101101010	w57	11010111011101110010101001100101
w26	10011000100011101111001000011110	w58	10010111100010010110011100100000
w27	10101010100100000001110010110111	w59	11111001110110010000111011101100
w28	01100100000010001011111111010011	w60	10010011011101000011000011001101
w29	01111010100111111100011011001001	w61	10110011001101011001100011111101
w30	10101001100010100011110100001100	w62	01100001100110000000000111000011
w31	01011111011110100010011101101111	w63	11101010010010001100111010101001

We can still see our original message, the appended '1' and the message length in the blocks. The difference this time around is that w^{16} ... w^{63} has been padded with information based on the calculations before.

4.3.5. Compression

We need to initialize variables a,b,c,d,e,f,g, and h and set them equal to the hash values respectfully from before.

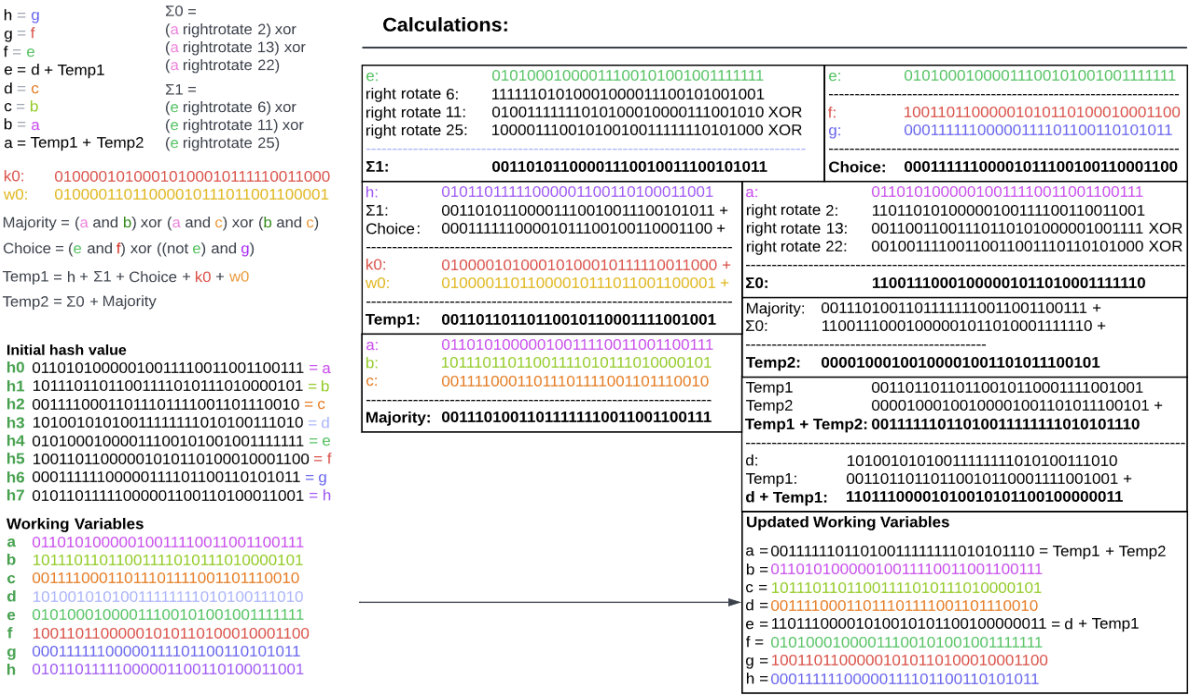


Figure 22: Calculation steps

These steps again are repeated until k^0 and w^0 are both k^{63} and w^{63}. Once the final compression is done, we can look to add the working variables to the current hash values to get our final output.

4.3.6. Modify Final Values

After the compression is done, we can modify the hash values as follows:

$$h0 = h0 + a$$
$$h1 = h1 + b$$
$$h2 = h2 + c$$
$$h3 = h3 + d$$
$$h4 = h4 + e$$
$$h5 = h5 + f$$
$$h6 = h6 + g$$
$$h7 = h7 + h$$

Figure 23: Modifying hash values.

4.3.7. Concatenate Final Hash

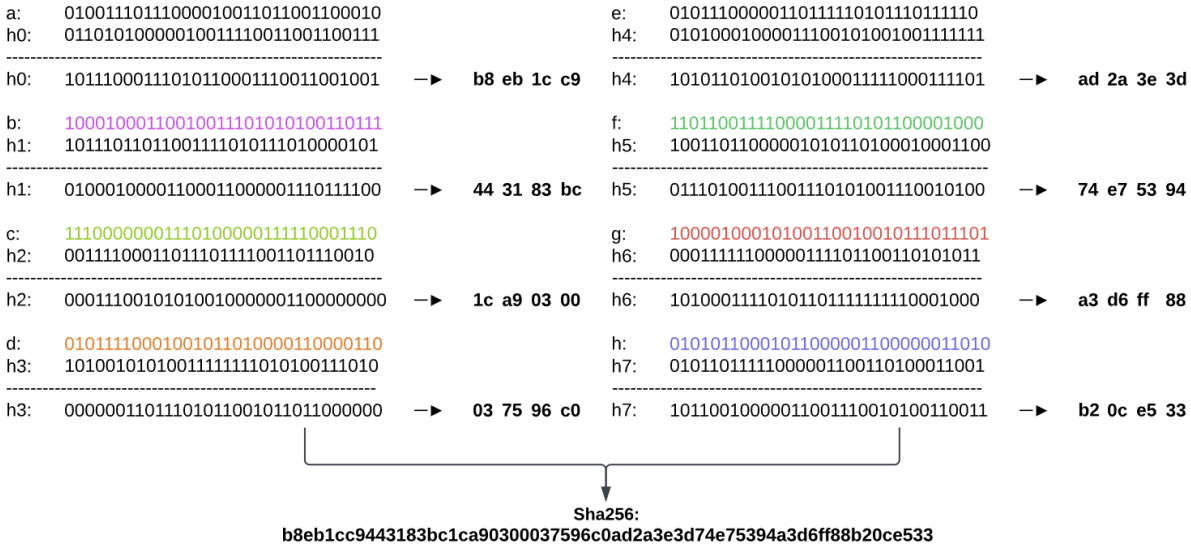


Figure 24: Calculating Hashes and combining them to make output.

Now that we have the final hash, we are done with the process. All steps were recreated using the SHA256 algorithm site (Martin, 2022).

4.4. SHA-3

As suggested by its name, SHA-3 is the successor to SHA-2. However, unlike SHA-2, SHA-3 uses a technique known as Sponge construction. It is important to understand how the sponge's function works. Below is an image of the process followed by the steps used in SHA-3 (Jon, 2021).

4.4.1. Sponge Function

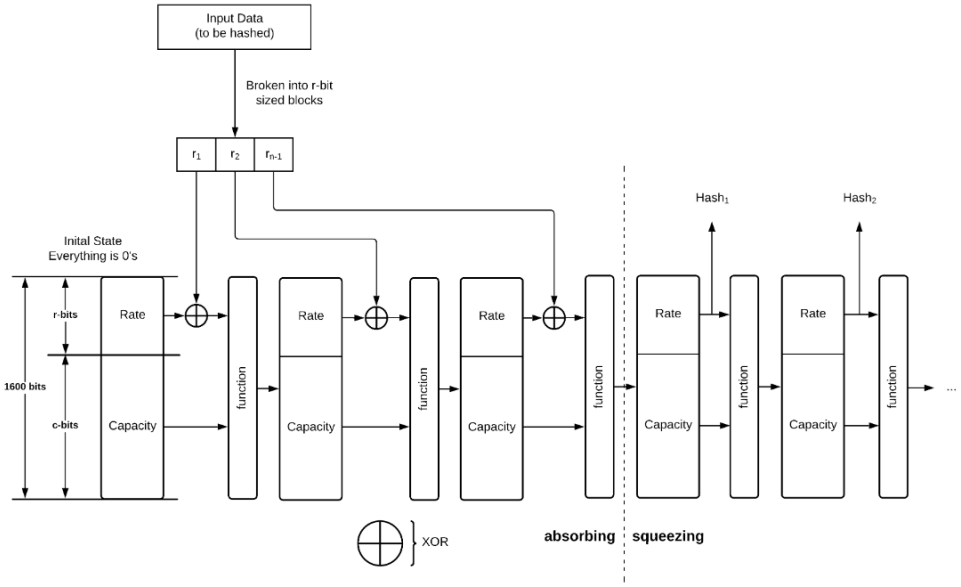


Figure 25: SHA-3 sponge function.

1. Split the plaintext data unto r-bit sized blocks, which with SHA3 is the rate + capacity bits sum up to 1600 bits.
2. The first input rate and capacity are zeros and XORed with the first-rate block r1.
3. The combined rate and capacity block is put through a function of multiple rounds.
4. The first r-bit of the function is the rate and reaming c-bit capacity.
5. The above r-bits are XORed with r2 and fed into another function.
6. This is done until no data blocks are left over.
7. On the last data block, the task is taken from the r-bits output of the function.
8. If more bits are needed, the rate and capacity are fed through the function without any more data.

Now that we understand how a basic sponge function works, we can begin to explain the SHA-3 hashing process.

4.4.2. SHA-3 Hashing

1. ***Padding*** – SHA-3 requires an equal number of perfect blocks to work, so it must pad the input to be perfectly divisible by r-bits. This is done by adding a 1 bit then filling in with zeros and ending again with a 1 bit.

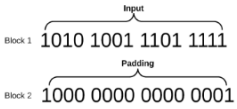


Figure 26: SHA-3 padding.

NIST has published a table of the four SHA-3 function standards, where we can see the values for rate r and capacity c.

Instance	Output size *d*	Rate *r* = block size	Capacity *c*	Definition	Security strengths in bits		
					Collision	Preimage	2nd preimage
SHA3-224(*M*)	224	1152	448	Keccak[448](*M* ‖ 01, 224)	112	224	224
SHA3-256(*M*)	256	1088	512	Keccak[512](*M* ‖ 01, 256)	128	256	256
SHA3-384(*M*)	384	832	768	Keccak[768](*M* ‖ 01, 384)	192	384	384
SHA3-512(*M*)	512	576	1024	Keccak[1024](*M* ‖ 01, 512)	256	512	512

Figure 27: NIST SHA-3 standard table

2. ***Block Transformation*** – The block transformation, which in the above sponge construction is labelled ‘Function’ is broken up into five steps done over several rounds. The state can be considered a 5 x 5 x w array of bits. The five steps are θ (theta), ρ (rho), π (pi), χ (chi) and ι (iota). These steps will be completed twenty-four times, which is the number of rounds SHA-3 uses.

3. **Step One: The θ (theta) Step**
   The θ step consists of two separate functions, the C function, and the D function.

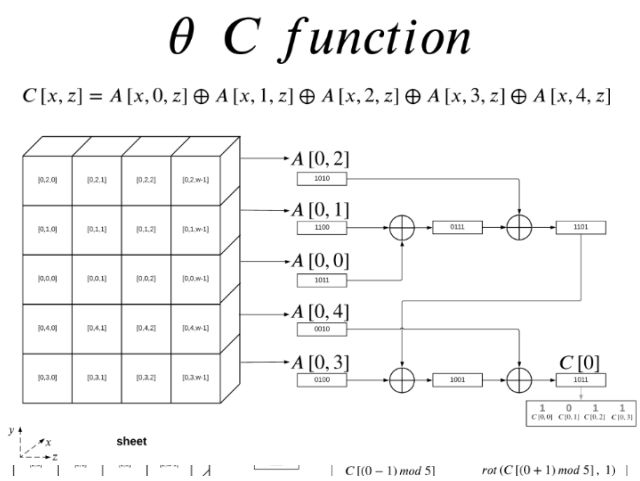


Figure 28: SHA-3 C function.

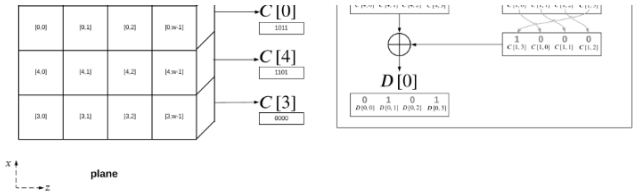


Figure 29: SHA-3 D Function

1. *C*alculate the column parities (C) – For each of the five columns in the state array, XOR all the bits in the same column across all rows to obtain a parity bit. Parity meaning is the property of the value is even or odd, this helps determine if the level of diffusion is sufficient. This results in a 1x5 array of parity bits denoted as C[0], C[1], C[2], C[3], C[4].

2. XOR neighbouring parties to create the D array. For each column, I, XOR is the parity of the adjacent columns. This generates a new 1x5 array Dm where D[i] = C[(i – 1) mod 5] ⊕ C[(i + 1) mod 5].

3. Update the state array. XOR each bit in the state array with the corresponding bit in the new 1x5 array D.

The theta step ensures a high level of diffusion, making it hard for an attacker to find patterns or relationships.

Step Two: The *ρ (rho) Step*

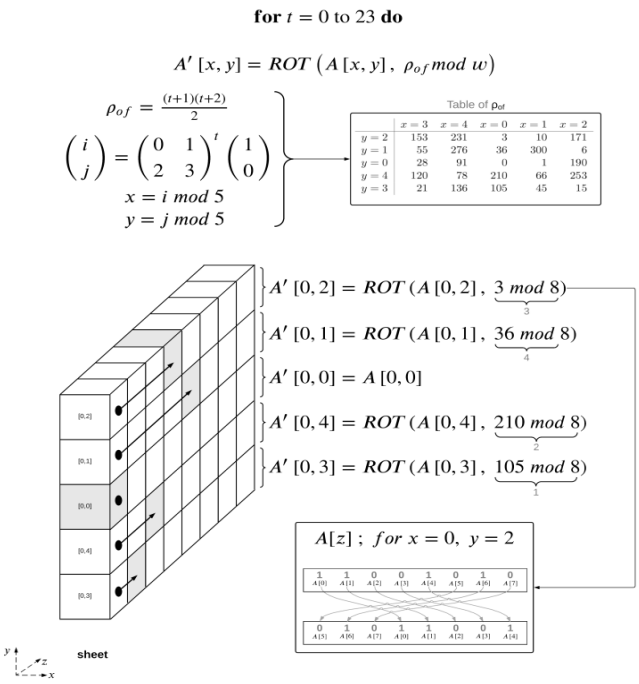


Figure 30:SHA-3 rho function.

1. For each lane (i, j) in the state array, rotate the bits of the lane by a fixed predetermined offset (r[i[[j]). This offset is determined by the following rules.

   a. For the lane at position (0, 0), the offset is 0.
   b. For the lane at position (i, j), the offset is determined by the formula: r[i][j] = (I * (I + 1) // 2) + (j*(2 * I + 3 * j) // 2) mod w (lane size).

   The lane size, w, depends on the hash lengths. W = 64 for SHA3-224, w = 32 for SHA3-384 and w = 16 for SHA3-512.

By rotating bits in each lane by these offsets, the Rho step provides additional diffusion within the state array making it more difficult for an attacker.

Step Three: The *π (pi) step*

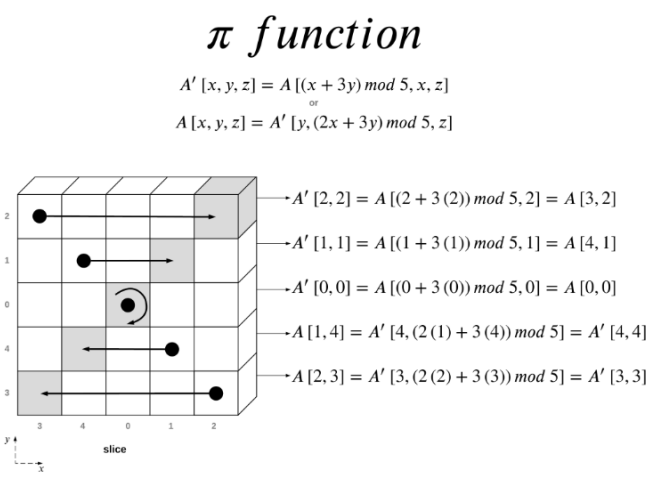


Figure 31: SHA-3 pi function.

1. For each lane (x, y) in the state array, move the lane to a new position (x', y') according to the following rules:

   (a) x' = y
   (b) y' = (2 * x + 3 * y) modulo 5.

   The new position (x', y') for a lane (x, y) is determined by swapping the x and y coordinates and applying a modular operation with the factor 5.

2. Update the state array with the new positions of the lanes.

By rearranging the lanes within the state array, the Pi step provides additional diffusion across the entire array.

Step Four: The *χ (chi) Step*

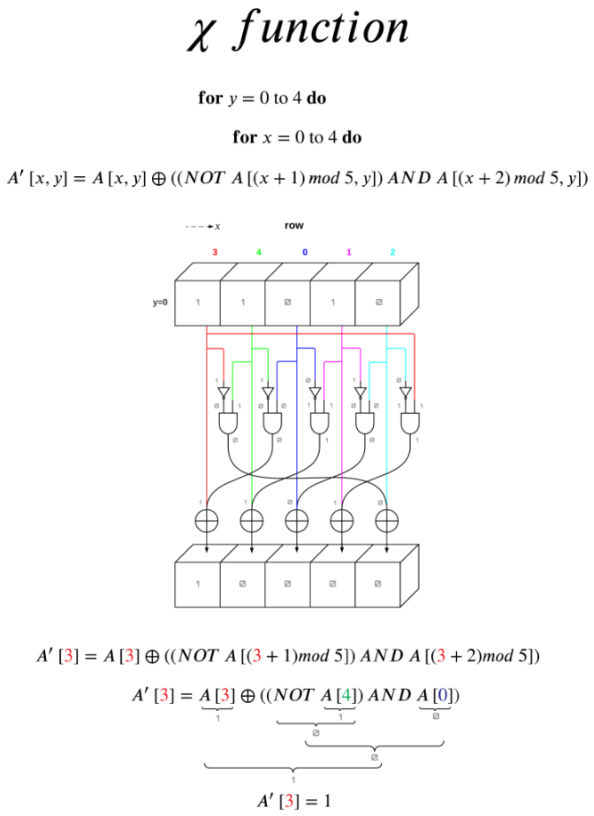


Figure 32: SHA-3 chi step

1. For each row in the state array, apply the following non-linear function on the lanes: A'[x,y] = A[x,y] ⊕ ((¬A[(x+1)%5,y]) ∧ A[(x+2) % 5,y])

   A[x,y] represents the original lane at position (x,y). A'[x,y] represents the updated lane value. ⊕ denotes the XOR operation, ¬ denotes the bitwise NOT operator and ∧ denotes the bitwise AND operator.

2. Update the state array with the new lane values.

The chi step adds non-linearity into the state array, which is essential for the hash functions' security, making it difficult for attackers to find patterns or relationships.

Step Five: The *ι (iota) Step*

$$\iota \; function$$

$$A'[x, y, z] = A[x, y, z] \oplus RC[i_r]$$

$$RC[i_r][0][0]\left[2^j - 1\right] \; for \; all \; 0 \le j \le \ell$$

j	[z]
0	0
1	1
2	3
3	7
4	15
5	31
6	63

$\ell = 6$

The Round Constant can only set 7 bits

This function only changes these 7 bits in the center lane

Figure 33: SHA-3 iota function

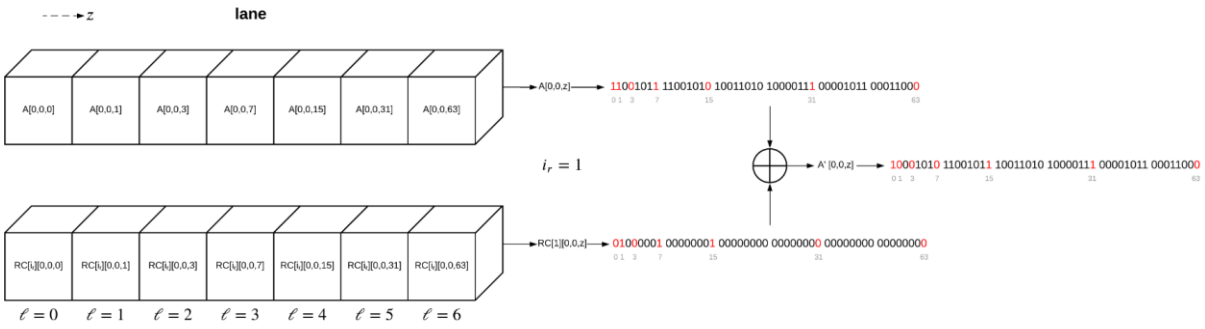


Figure 34: SHA-3 Iota function 2

1. XOR the lane position (0,0) in the state array with a round constant (RC). The round constant is different for each round and is derived from a predefined set of constants.
2. Update the state array with the modified lane value.

The round constants are generated using a linear feedback shift register (LFSR) and are designed to introduce diffusion.

RC[00] = 0x0000000000000001	RC[12] = 0x000000008000808B
RC[01] = 0x0000000000008082	RC[13] = 0x800000000000008B
RC[02] = 0x800000000000808A	RC[14] = 0x8000000000008089
RC[03] = 0x8000000080008000	RC[15] = 0x8000000000008003
RC[04] = 0x000000000000808B	RC[16] = 0x8000000000008002
RC[05] = 0x0000000080000001	RC[17] = 0x8000000000000080
RC[06] = 0x8000000080008081	RC[18] = 0x000000000000800A
RC[07] = 0x8000000000008009	RC[19] = 0x800000008000000A
RC[08] = 0x000000000000008A	RC[20] = 0x8000000080008081
RC[09] = 0x0000000000000088	RC[21] = 0x0000000080008080
RC[10] = 0x0000000080008009	RC[22] = 0x8000000000000080
RC[11] = 0x000000008000000A	RC[23] = 0x8000000000008008

Figure 35: Round Constants for Iota step

5. TwoFish

TwoFish is a symmetric key block cipher with a block size of 128 bits and a key size of up to 256 bits. (“Twofish - Wikipedia”) TwoFish is known for its distinctive features such as key-dependant S-boxes and a complex key schedule. TwoFish uses a key block structure called the **Feistel Scheme**, which includes splitting the plaintext into two blocks and running them through different function rounds and then finally XORing them and swapping the two halves to get your ciphertext/
Below is an image of the TwoFish Feistel scheme (Khan, n.d.).

5.4.1. TwoFish Key Generation

Like AES, TwoFish also uses an S-box, but with a twist.

1. **S-Boxes** - The S-box is a key-dependent S-box, meaning each S-box is defined with 2-4 bytes of key material depending on the key size. This adds extra security to the S-box as it can be of different sizes. An example of the TwoFish 128-bit key is shown below:

$$
\begin{aligned}
s_0(x) &= q_1[q_0[q_0[x] \oplus s_{0,0}] \oplus s_{1,0}] \\
s_1(x) &= q_0[q_0[q_1[x] \oplus s_{0,1}] \oplus s_{1,1}] \\
s_2(x) &= q_1[q_1[q_0[x] \oplus s_{0,2}] \oplus s_{1,2}] \\
s_3(x) &= q_0[q_1[q_1[x] \oplus s_{0,3}] \oplus s_{1,3}]
\end{aligned}
$$

Figure 36: TwoFish 128-bit S-Box

Where $s_{i,j}$ are key bytes derived from the key bytes? Even with equal key bytes, the S-boxes will still differ from one another.

2. **Maximum Distance Separable (MDS)** - Once the output of the S-boxes is computed, they are multiplied by the MDS matrix. The MDS matrix is always the following:

$$
\begin{bmatrix}
01 & EF & 5B & 5B \\
5B & EF & EF & 01 \\
EF & 5B & 01 & EF \\
EF & 01 & EF & 5B
\end{bmatrix}
$$

Figure 37: MDS matrix

The MDS must have the following properties:

1. **Maximum Distance** – It maintains the maximum possible Hamming distance between the input and output pairs, meaning that on average, half of the output buts change if a single bit is changed, increasing its complexity.

2. **Invertibility –** The MDS matrix is invertible, which is important to decrypt. The inverse matrix is used in decryption to reverse the diffusion effect.
3. **Key Expansion** – Key expansion is the process of deriving a set of round keys from the initial encryption key. These rounds will be used later in encryption and decryption. TwoFish uses a total of forty rounds, eight for input and output whitening and thirty-two for the sixteen rounds of the F function which will be explained later. There are multiple stages to key expansion as follows.

   (a) Convert the input key into a series of 32-but words. Divide the input key into K[0], K[1] ... K[7].

   (b) Calculate the Reed-Solomon (RS) matrix, which is shown below.

$$\begin{bmatrix} 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 \\ 01 & 02 & 04 & 08 & 10 & 20 & 40 & 80 \\ 01 & 04 & 10 & 40 & 02 & 08 & 20 & 80 \\ 01 & 08 & 40 & 02 & 04 & 10 & 20 & 80 \end{bmatrix}$$

Figure 38: Reed-Solomon matrix

   (c) Generate the round key constraints. These are derived from the RS matrix and denoted as Me and Mo. These are generated by multiplying the input key K[] by the RS matrix as follows.

$$\begin{bmatrix} Me = [K[0]\ K[2]\ K[4]\ K[6]] \\ \\ Mo = [K[1]\ K[3]\ K[5]\ K[7]] \end{bmatrix}$$

Figure 39: Generating round key constraints with RS matrix.

(d) Compute the round subkeys – These are generated using the key-dependant S-boxes, the h() function and the round key constraints Me and Mo. The h() function takes a 32-bit input and produces a 32-bit output. The formula for doing so is shown below

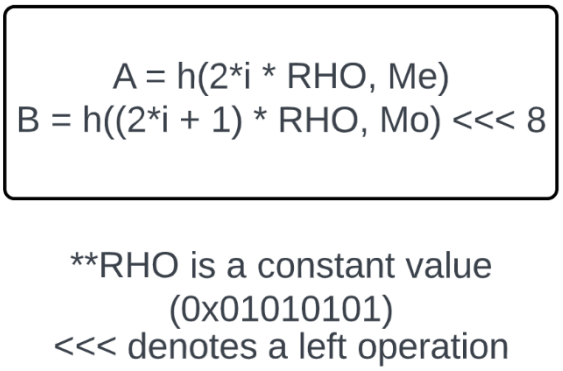


Figure 40: Computing round subkeys

(e) Organise the round subkeys. They are organised into the following groups.
i. The first eight subkeys are used for input and output whitening.
ii. The remaining 32 subkeys are used for the 16 rounds of the F function.

5.4.2. TwoFish Encryption

There are four main steps to TwoFish encryption, and they are as follows.

1. **Input Whitening –** This is when you XOR the input plaintext block with the first four round keys so that W0 = P0 XOR K0. The resulting blocks W0 etc, are then fed into the sixteen rounds of the F function. Input whitening ensures that the plaintext data is mixed with the key contents before entering the main rounds of the scheme increasing its security.
2. **F function –** The F function is a non-linear operation during each round of encryption and decryption. It consists of sixteen rounds, and it introduces confusion making it difficult to attack. Each round of the F function consists of the following steps.
   i. **Key mixing –** At the start of every round, the input block is XORed with the round key. There are two 32-bit round keys per round. The round key mixing ensures that the round keys are integrated into the internal encryption process.
   ii. **Substitution -** The input block is divided into 8-bit blocks, and each block is substituted with the corresponding value from the key-dependant S-box.
   iii. **Permutation –** The outputs from the S-boxes are multiped by the MDS matrix, creating the diffusion effect, where one-bit results in a change in every other bit.
   iv. **Pseudo-Hadamard Transform (PHT) –** The outputs from the MDS matrix are combined using a PHT operation. The PHT is a linear operation that mixes data from the two 32-bit halves of the input block. The PHT is defined as:

$$A' = A + B \bmod 2^{32}$$
$$B' = A + 2*B \bmod 2^{32}$$

Figure 41: Pseudo-Hadamard Transform

Where A and B are the 32-bit input halves and A', and B' are the 32-bit output halves.

v. **Output Whitening –** After the 16 rounds of the F function, you will have output blocks of four 32-bit words R0, R1, R2 and R3. XOR each one with the corresponding round key to get the resulting ciphertext blocks, C0, C1, C2, and C3.

5.4.3. TwoFish Decryption

Since TwoFish is a symmetric system, the decryption process is essentially the same as encryption, except in reverse order and using the inverse of the matrixes. Below is a summary of the decryption process.

1. **Output Whitening –** XOR the ciphertext with the output whitening round keys to reverse the output whitening.
2. **F Function –** Perform the sixteen rounds of the F function in reverse order. Apply the inverse of the PHT and multiply the outputs by the inverse MDS matrix. Then use the inverse of the key-dependant S-boxes and use the inverse order of the round keys.
3. **Input Whitening –** XOR the output block from the F function with the input whitening round keys to reverse the input whitening step and retrieve the original plaintext.

TwoFish is known for its high efficiency, flexibility, and availability as it's open source. It is often used for VPNs, password storage and even software tamper protection.
TwoFish does have its issue though such as its complexity, making it harder to implement as well as its need for a larger key length than AES to achieve the same level of security.

6. What are quantum algorithms?

A quantum algorithm does the same procedure as classical algorithms where it is a step-by-step procedure to perform instructions to solve a problem. Technically, all pre-quantum algorithms can be performed on a quantum computer. They are not considered quantum as they do not involve these two quantum-specific steps (a) **superposition** and (b) **entanglement.**

a) **Superposition** separates a quantum bin or qubit from a classical bit. A classical bit can only be one state at a time, either '0' or '1'. A measurement of a classical bit would not disrupt its state. A quantum bit can be a superposition, meaning it can be both states '0' and '1'. A measurement of this qubit would destroy its coherence and disrupt its superposition state, effectively making it choose '0' or '1' (Metwalli, 2023).
b) **Entanglement** is when a pair or group of particles are entangled, and the quantum state of each particle cannot be described as independent of the other particle(s). ("Superposition and entanglement - Quantum Inspire") If the bit in one particle is '1', so will the remaining particles, even if separated by a large distance. Entanglement allows superdense coding, which is a quantum communication protocol to communicate the number of classical bits by transmitting a small number of qubits, under the assumption the sender and receiver pre-share an entangled resource. We can send 2 bits via a single qubit and
can choose four quantum gate operations (Emspak, 2022).

6.5. How does Shor's algorithm work?

Shor's algorithm makes use of both classical and quantum methods to find the main integer N (Rakhade, 2020). To find N, we need to know the two prime factors, g, and N.
We can use Euclid's algorithm, which helps us determine the greatest common factor (GCF). To use this, we subtract the smaller factor from the bigger number. Repeat this until one of the factors is 0. For example, to calculate the GCF of 42 and 105 we can do the following:

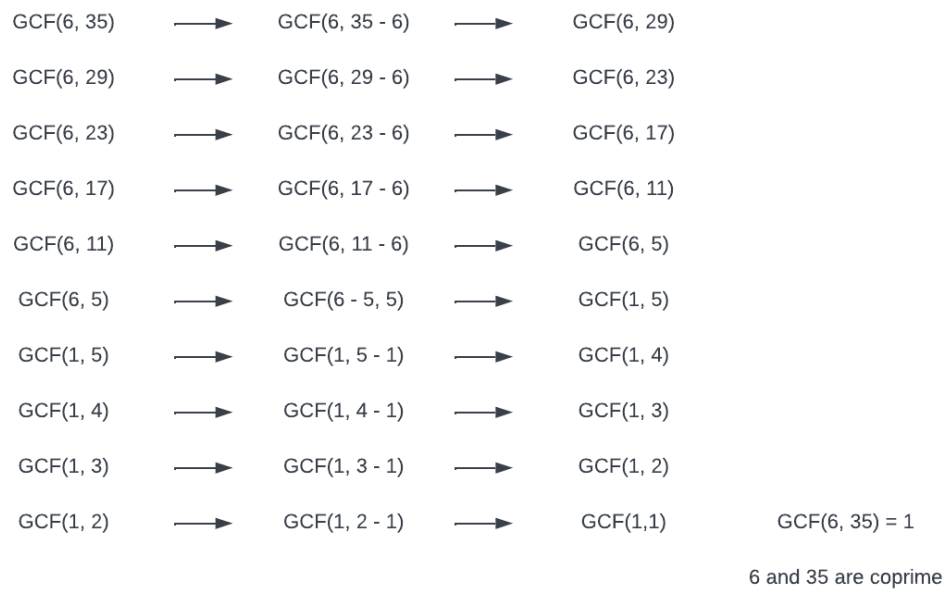


Figure 42: Euclid's Algorithm to get GCF.

6.5.1. Finding Factors of N

We do not need to guess a direct factor, rather a value that shares a factor is enough.
We can check GCF(g, N) with Euclid's Algorithm. If the GCF(g, N) does not equal one, a factor of N was found, and we can calculate N using g and N/g.
If the GCF(g, N) == 1, we have found the co-primes, this means we need a better guess, and this is the more likely outcome. This is where we utilize Shor's algorithm as it helps us obtain a better guess.
If two values do not share factors and GCF(X, Y) == 1, then we can take our two co-primes, X and Y,

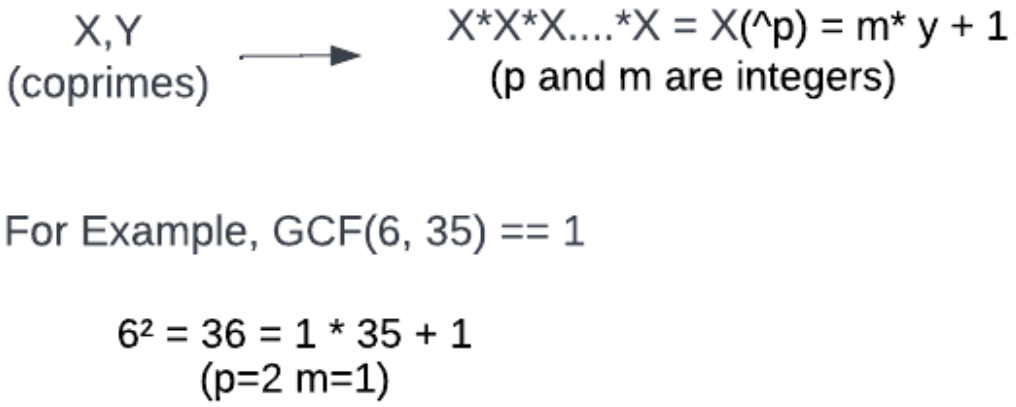


Figure 43: Example is given p to show how the equation relates to co-primes.

6.5.2. What can we do with this?

Using our early logic and math, we can improve our guess, g to find a factor of N:

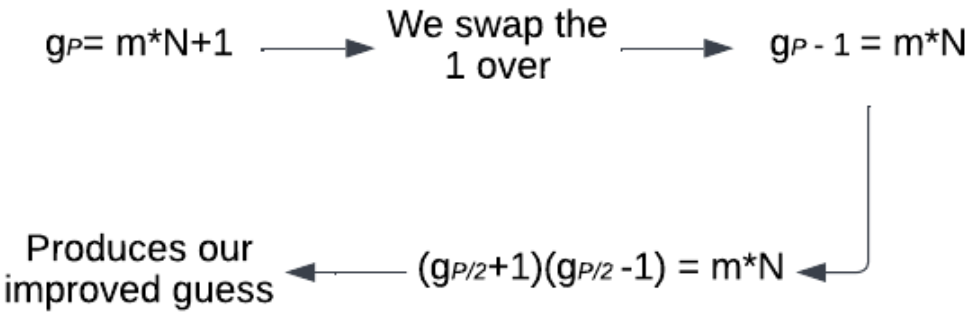


Figure 44: How Shor's algorithm improves our guess, g

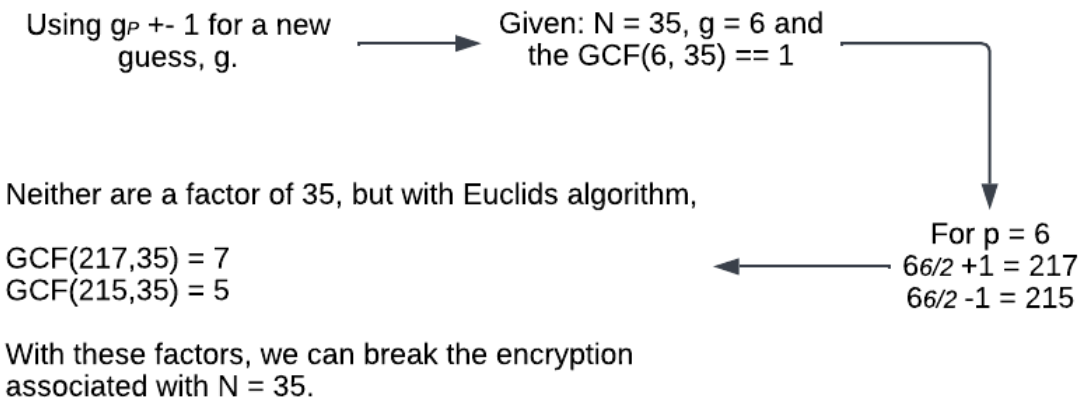


Figure 45: Using Euclid's algorithm to solve for N.

6.5.3. Where are the Quantum aspects?

We used small integers and primes in our example, which is not very demanding on computational resources, so is easy to do. The trouble comes in when you are using large numbers, it takes a lot of time and resources to find p. Not every p will work and provides some issues such as.

a) We don't want a p-value that produces a multiple of N, we just want factors of N.
b) Odd numbers produce numbers with fractions which are hard and expensive to deal with.
c) As N gets larger, search space for p increases drastically, testing so many values of p would take too long.

This is where quantum computing comes in. When a quantum computer runs Shor's algorithm, it combines superimposed possible answers with interference will allow us to compute the answer p, in a single computation.

7. Post-Quantum Cryptography

7.6. Definitions

Polynomials – A mathematical expression that consists of variables and coefficients, combined using addition, subtraction, multiplication, and non-negative integer exponents (Mathsisfun, n.d.).

(Levasseur, 2021)- Is a mathematical structure that consists of a set of polynomials with coefficients from a given field.

Coefficients - The coefficients of a polynomial are the constants that are multiplied by the variables in each term. For example, in the polynomial 2x^3 - 5x^2 + 3x + 7, the coefficients are 2, -5, 3, and 7.

Irreducible - An irreducible polynomial is a polynomial that cannot be factored into a product of two or more non-constant polynomials with coefficients from the same field (Joyce, 2016).

Matrix - A matrix is a rectangular array of numbers or symbols that can be used to represent a system of linear equations, transformations, or data.

Noise - Refers to errors or disturbances that can occur during the transmission or processing of data. Often used to deter eavesdroppers (University of Basel, 2020).

Error Matrix - A matrix that represents the errors or noise that occur during the transmission or processing of data.

Euclidean Norm - A mathematical concept that measures the length or magnitude of a vector in a Euclidean space.

Parity Check - A mathematical concept that measures the length or magnitude of a vector in a Euclidean space.

Random Oracles – This is a hypothetical mode in hash function proofs to represent a perfect hash function. The oracle believes that each unique input produces a truly random output that is uniformly distributed, these are used in cryptographic proofs to simplify and provide security guarantees, but in practice, they do not exist.

7.7. Lattice-Based Cryptography

Lattice-based cryptography is one of the most promising branches of post-quantum cryptography which focuses on developing quantum-resistant algorithms. Lattice-based cryptography is very efficient in providing secure encryption, digital signatures, and even key exchange protocols. As quantum computers approach, there is a need for quantum resistance, and Lattice-based cryptography is one of the strongest candidates to provide a secure future (Chi, 2015).

7.7.1. What is a lattice?

A lattice can be thought of as two points stretched out on an infinite field.

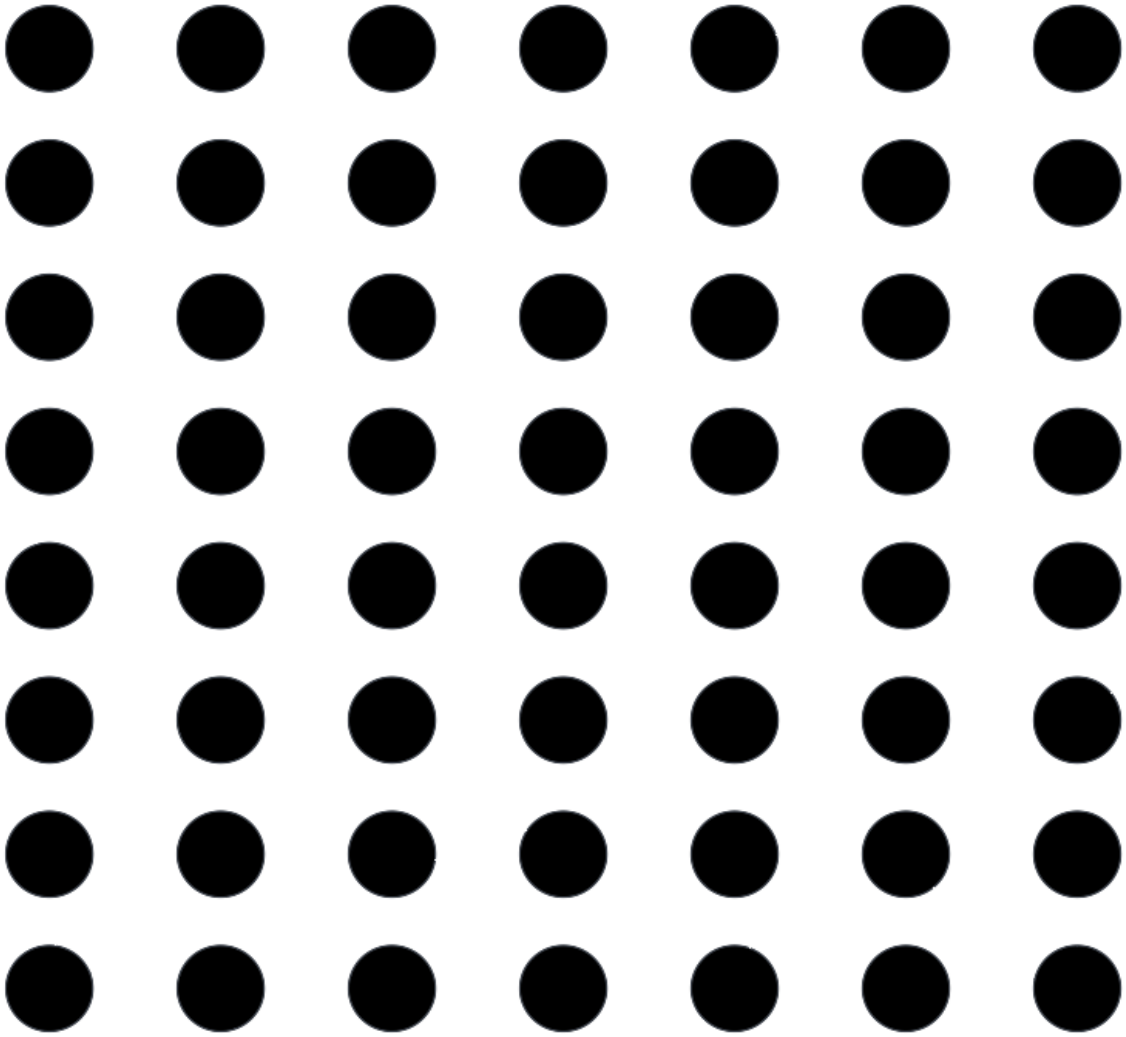


Figure 46: Lattice Example.

Every dot in the above is called a vector, so a vector is just a point in the lattice above. We can see that the vectors are spaced out above, which is considered a lattice, being a collection of evenly spaced vectors.

7.7.2. What is a basis of a lattice?

Lattices are infinitely sized objects, but computers only have a finite amount of memory to work with, so we need a method to represent lattices in a finite form. For this, we use a 'bases' of a lattice, meaning a small collection of vectors to reproduce any point in the grid that forms the lattice.

7.7.3. How is a basis calculated?

Take the lattice above and plot a point on the lattice above, making sure to select two points that do now lie on the same line. We can plot two points on (2,0) and (0,2) as follows.

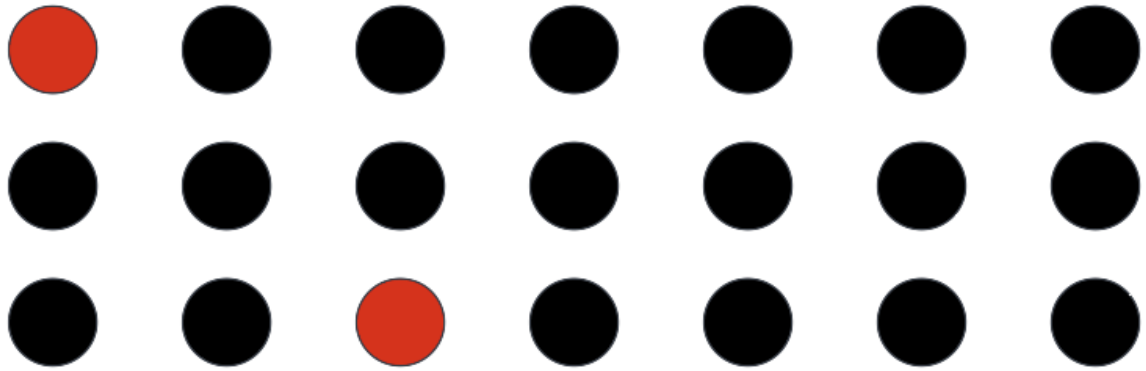


Figure 47: Plotted points on a lattice

We can take two integers, in this case, 3 and -1. We multiply the coordinates of the first point by 3, and the second point by -1, resulting in a new point (6, -2). We can keep doing this with our two original coordinates to draw a grid. The basis consisting of vectors (2,0) and (0,2) generates the lattice points with even coordinates. The idea is, by choosing a basis we have chosen an entire lattice. We have now just created a finite grid from an infinite field. We can have many different bases, by just choosing different starting coordinates. It's important to know that a shorter basis is better at solving lattice problems in cryptography than long ones. This is due to the computational efficiency with short basis over longer ones.

7.7.4. Lattice-based problems

There are multiple mathematical problems related to lattices, but we are most interested in the Learning With Errors (LWE) problem and its variants Ring-Learning With Errors (Ring-LWE), Learning With Errors Over Module Lattices (MWLE), the Short Integer Solution problem (SIS) and the Module Short Integer Solution problem (Module-SIS).

7.7.5. The Learning With Errors Problem

The LWE problem can be described as follows: Given the secret vector s ∈ Z_q^n and a random vector a ∈ Z_q^n, the LWE problem involves finding the secret vector s from a noisy linear equation: $t = a * s + e \pmod q$ where: t ∈ Z_q is the noisy inner product, e ∈ Z_q is a small error term, and q is a modulus. ∈ denoting 'is an element of'.
The LWE problem I believed to be hard because the presence of the error term e makes it computationally infeasible to recover the secret vector s. This is closely related to the hardness of other lattice problems such as the Shortest Vector Problem. The problem involves searching for the shortest non-zero vector in a lattice. Some algorithms compute an estimate of this, but none are 100% accurate. It's believed not even quantum computers can efficiently solve this issue. The Falcon post-quantum algorithm relies on the LWE problem.

7.7.6. The Ring Learning With Errors problem

Ring-LWE is a variant of LWE, but rather than operating around vectors, it operates round polynomial rings. Ring-LWE is more compact and efficient in cryptographic terms. The Ring-LWE problem can be described as follows: Given a secret polynomial 's' and a random polynomial 'a', both from polynomial ring R the Ring-LWE problem involves finding a secret polynomial s from a noise polynomial equation, $t = a * s + e \pmod q$, where t is the noisy product of the polynomials 'a' and 's', e is the small error polynomial and q as the

modulus.
The polynomial ring R is usually defined as R = Z_q[x] / (f(x)), where Z_q[x] is the ring of integer polynomials with coefficients modulo q, and f(x) is an irreducible polynomial.
The hardness of the Ring-LWE is based on the LWE but has more compact and efficient cryptographic schemes.

7.7.7. The Learning With Errors Over Module Lattices

This is another variant of the LWE problem but also combines the advantages of Ring-LWE and LWE. The MWLE problem can be described as follows: Given a secret matrix S ∈ Z_q^{n x m} and a random matrix A ∈ Z_q^{n x m}, the MLWE problem involves finding the secret matrix S from the noise equation: T = A * S + E (mod q) where, T ∈ Z_q^{n x m} is the noisy product of the matrices A and S, E ∈ Z_q^{n x m} is a small error matrix and q is a modulus.
The module lattice allows for more cryptographic constructions with varying levels of efficiency and security. Both CRYSTALS-Kyber and CRYSTALS-Dilithium rely on the MWLE.

7.7.8. The Short Integer Solution problem

The Short Integer Solution (SIS) is closely related to the LWE problems and can be described as follows: Given a random matrix A ∈ Z_q^{n x m} (with n < m), and a positive integer bound β, the SIS problem involves finding a non-zero vector x ∈ Z^m such that:
A * x = 0 (mod q) and ||x|| ≤ β where A * x denotes the matrix-vector multiplication, ||x|| denotes the Euclidean norm (or length) of vector x and q is the modulus.
The aim is to find a short, non-zero integer vector x that satisfies the linear equation modulo q. SIS is considered hard due to it being computationally infeasible to find such a short vector when given the random matrix A. Some algorithms can make an estimated guess, but none can make a guarantee.

7.7.9. The Module Short Integer Solution Problem

The Module Short Integer Solution (Module-SIS) is a generalisation of the SIS problem, extending it to include module lattices. The Module-SIS problem can be described as follows: Given a random matrix A ∈ Z_q^{n x m}, an integer bound β, and a module structure defined by a polynomial ring R = Z_q[x]/(f(x)), the Module-SIS problem involves finding a non-zero vector x ∈ R^m such that: A * x = 0 (mod q) where A * x denotes the matrix-vector multiplication, with coefficients in the polynomial ring R, ||x|| denotes an appropriate norm (or length) of the vector x, typically defined over the coefficients of the polynomials, f(x) is an irreducible polynomial and q is the modulus.
The objective is to find a short, non-zero vector x with coefficients in the polynomial ring R that satisfies the linear equation modulo q.

7.7.10. Lattice-based Cryptography Conclusion

Lattice-based cryptography has already become the new direction of cryptography, as three of the four algorithms from the NIST Post-Quantum Cryptography Standardization competition were lattice-based algorithms. There are already many improvements made in the efficiency using the problems mentioned above and research is constantly being done to uncover more information.

7.8. Multivariate Quadratic Cryptography

Multivariate Quadratic (MQ) Cryptography relies on the hardness-solving systems of multivariate quadratic equations over finite fields. In MQ, the security is based on the difficulty of solving a system of n quadratic polynomial equations with m variables over a finite field F, typically named F_q (Huang, n.d.).

7.8.1. What is a multivariate quadratic equation?

A multivariate quadratic equation is an equation in which the highest degree of all variables is 2. In other words, it is a polynomial equation in which each term has at most two variables raised to a power of two. 4x^2 + 3xy + 2y^2 = 7 is considered a multivariate quadratic equation. Multivariate quadratic equations can have multiple variables and can involve cross-terms, i.e., terms involving products of two different variables.

7.8.2. What makes multivariate quadratic equations over finite fields difficult?

Exponential Growth – With m variables, the search space for solutions in a finite field grows exponentially with the number of variables, making it hard to brute-force solutions.

Non-Linearity - Quadratic equations introduce nonlinear relationships among the variables, making it more challenging to solve by making it more computationally complex.

Lack of Algorithms – There are no current algorithms that can currently solve instances of the MQ problem efficiently.

NP-Hardness - The MQ Problem is known to be NP-hard under certain assumptions, which indicates that it is unlikely to have an efficient algorithm for solving it in the general case.

MQ algorithms consist of a public key, which is a system of multivariate quadratic equations, and a private key. The most popular application of MQ cryptography is in the design of digital signatures such as the Unbalanced Oil and Vinegar (UOV) and the Rainbow scheme. The Rainbow scheme is a generalisation of the UOV and was entered into the previously mentioned NIST post-quantum competition.

7.8.3. The Unbalanced Oil and Vinegar (UOV) scheme.

The UOV scheme is a digital signature scheme based on Multivariate Quadratic (MQ) Cryptography. The idea is to construct a trapdoor function based on systems of multivariate quadratic equations. The variables are partitioned into two sets: oil variables and vinegar variables. The number of vinegar variables is usually larger than the number of oil variables, hence the name "unbalanced." The UOV scheme consists of three main algorithms:
Key Generation – A private and public key pair is created. The public key is obtained by composing the private key's quadratic equations with the affine transformations.

Signing - The signer computes the hash of the message and then solves the private key's quadratic system using the easily solvable structure. This results in a signature that is a vector of values for the oil and vinegar variables.

Verification - The verifier first computes the hash of the message and then substitutes the signature's values for the oil and vinegar variables in the public key's quadratic equations.

The security of the UOV scheme relies on the hardness of solving the system of multivariate quadratic equations (the MQ Problem) without knowledge of the trapdoor information (the

private key). As mentioned before, the Rainbow post-quantum algorithm is based on UOV and will be explained in greater detail in this document.

7.9. Code-Based Cryptography

Code-Based Cryptography relies on the hardness of decoding random linear codes (Singh, 2019).

7.9.1. The Decoding problem

The hardness of decoding random linear codes forms the foundation of code-based cryptography. The most well-known problem in this area is the "decoding problem," which can be described as follows: Given a linear code, a received word, and a parameter t (number of errors), the goal is to find the original code word that is at most t errors away from the received word. This problem is NP-hard, which indicates that it is unlikely to have an efficient algorithm for solving it in the general case.

7.9.2. Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC)

QC-MDPC codes are a specific class of error-correcting codes which are efficient and have small key sizes. QC-MDPC codes are a variant of Low-Density Parity-Check (LDPC) codes with a quasi-cyclic structure, which allows for more compact representations and efficient implementations.

In QC-MDPC codes, parity check matrices are constructed using cyclic shifts of a small number of fixed submatrices, which are typically square matrices of size 4x4 or 8x8. These submatrices are designed to have good error-correcting properties, and the cyclic shifts allow for efficient encoding and decoding of the codes. QC-MDPC codes are characterized by their rate, which is the ratio of the number of information bits to the total number of bits in the code. The rate of QC-MDPC codes can be adjusted to achieve different levels of performance.

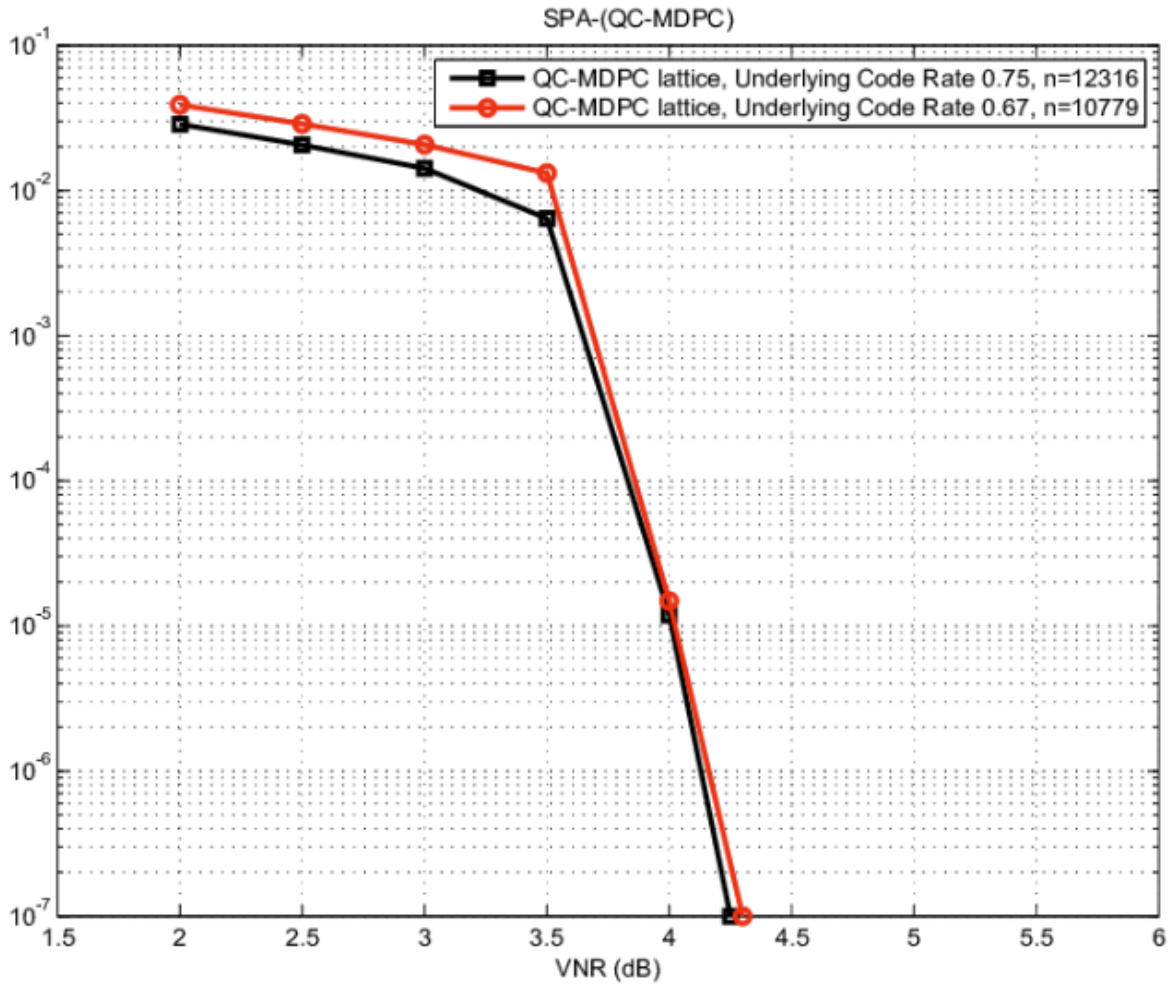


Figure 48: The error performances of two QC-MDPC lattices with n 0 = 3 (rate 0.67) and n 0 = 4 (rate 0.75)

7.9.3. What makes QC-MDPC hard to solve?

Large Search Space - When decoding a random linear code, the number of possible codewords grows exponentially with the code's length.

Error Correction - The error correction capability of a code is the maximum number of errors that can be corrected. For QC-MDPC codes, this capability is moderate, meaning that a significant number of errors can be corrected. However, the decoding problem becomes harder as the number of errors increases, approaching the code's error correction capability. It also becomes more resource-consuming to continue error corrections.

Decoding Algorithms - Regarding random linear codes and QC-MDPC codes, no efficient decoding algorithms are known for the general case, and the best-known algorithms have exponential complexity in the worst case.

NP-Hardness - While the hardness of decoding QC-MDPC codes specifically has not been proven to be NP-hard, the problem is still believed to be hard for both classical and quantum computers.

The quantum-resistant algorithm BIKE uses QC-MDPC in its systems.

7.10. Hash-Based Cryptography

As previously mentioned, hashing functions do not rely on the mathematical problems that the symmetric and asymmetric key systems in classical cryptography. This means that hashing is somewhat quantum-resistant. Some care does need to be taken as hash functions using smaller key sizes may be broken at an accelerated speed, but it is still believed that algorithms like SHA-256 and SHA-3 will be quantum-resistant. This, however, does not make them quantum algorithms as they do not hold any quantum properties as listed previously. The final finalist of four was SPHINCS+ which is a hash-based quantum algorithm. SPHINCS makes use of both hash functions and Merkle Trees for its security and performance.

7.11. Non-Interactive Zero-Knowledge Proofs (NIZPKs)

7.11.1. What is a Zero-Knowledge Proof?

A standard zero-knowledge proof is a way of proving the validity of a statement without revealing the statement itself. The 'prover' is the party trying to prove a claim, while the 'verifier' is responsible for validating the claim. The communication is in real-time and interactive. This is a big flaw as it can often be slow and tedious to complete (Ethereum, 2022).

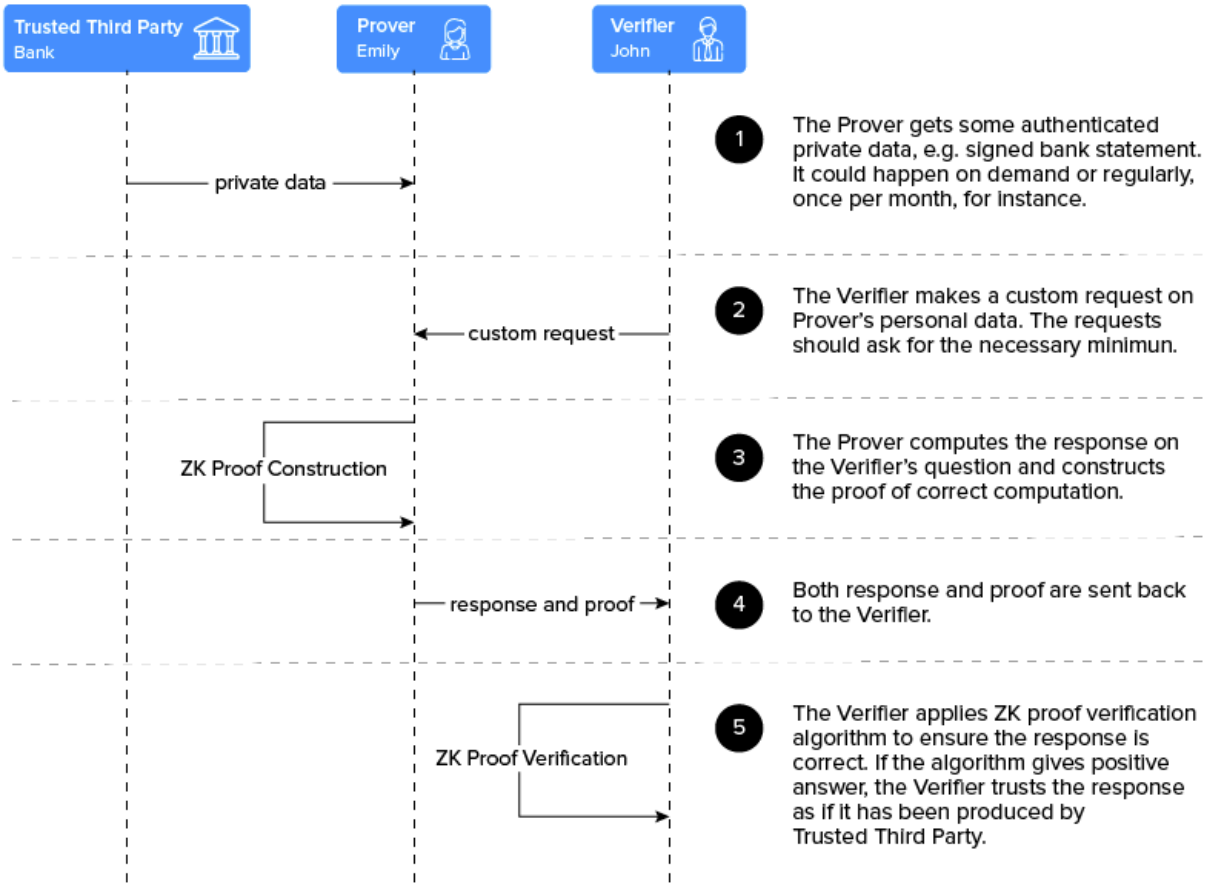


Figure 49: Zero-Knowledge Proof Example

7.11.2. What is a Non-Interactive Zero-Knowledge Proof?

The prover creates a proof without any interaction with the verifier. This proof can then be verified by anyone with access to the public information and the proof itself. The prover creates a proof without any interaction with the verifier. This proof can then be verified by anyone with access to the public information and the proof itself. If the statement is true and both the prover and verifier follow the protocol, the verifier will accept the proof as valid (Geeksforgeeks, 2022).

7.11.3. How does the process change from a ZKP to a NIZKP?

A common technique is the Fiat-Shamir heuristic, which transforms a ZKP into a NIZKP. This involves replacing the verifier's random challenges with a deterministic function, usually a hash function, of the prover's commitments and public information. This allows the prover to generate the proof independently, but it requires the assumption that the hash function used is secure and behaves like a random oracle.

7.12. CRYSTALS-Kyber

In the following paragraphs, for any terms underlined and highlighted in **bold, see** the end of this section for definitions.

CRYSTALS-Kyber is a post-quantum key encapsulation mechanism that bases its security on the learning with errors over module lattices problem which combines the usual LWE with the ring variant problem (Ring-LWE), It is believed to be resistant to both classical and quantum attacks (Roberto Avanzi, 2021). It is designed to provide a secure and efficient key exchange for use in a variety of applications such as secure communication, digital signatures and more.

Kyber was selected by NIST in July 2022 as one of the four winners of their Post-Quantum Standardisation (NIST, 2022).

7.12.1. Key Generation

In this step, a public-private key pair is generated for the receiver.

1. **Choose Parameters** - Select the parameters for the scheme, including modulus q, the **ring dimension** n and the **error distribution** ψ. In Kyber, n is a power of 2, and ψ is typically a discrete **Gaussian distribution** or **binomial distribution.**
2. **Define the Ring R_q** – Construct the polynomial quotient ring R_q = Z_q[X] / (Φ_n(X)), where Φ_n(X) = X^n + 1 is the n-th **cyclotomic polynomial**. Polynomials in R_q have a degree less than n, and their **coefficients** are elements of the finite field Z_q.
3. **Sample a Random Polynomial 'a** '– Choose a **uniformly random** polynomial 'a' from R_q. This polynomial serves as a public parameter as is part of the receiver's public key.
4. **Sample Secret and Error Polynomials –** Choose two polynomials 's' and 'e' from the error distribution ψ. The polynomial 's's servers as the receiver's secret key, while 'e' is used to add noise to the public key, ensuring the hardness of the underlying problem.
5. **Compute the public polynomial 'b' –** Calculate the public polynomial 'b' b = a * s + e. The polynomial 'b' is computed in the ring R_q, which means the arithmetic is performed modulo q and modulo the **ideal generated** Φ_n(X).

6. **Public and secret keys –** The public key is the pair (a,b) and the secret key is the polynomial ‘s’. The public key can be shared with others who want to securely communicate with the receiver, but the secret key must be kept private.

The whole process relies on the hardness of the ring-LWE problem, which makes it hard for any attacker to recover the secret key from the public key even with a quantum computer.

7.12.2. Encapsulation

Kyber allows a sender and recipient to securely transmit a shared secret to begin communication. To encapsulate a key with Kyber, a sender generates a random symmetric encryption key and uses the recipient’s public key to encrypt it. The encrypted key is then sent to the recipient who uses their secret key to decapsulate the shared secret to retrieve the symmetric encryption key for an encryption algorithm such as AES. To begin the encapsulation process you must do the following. To clarify, steps 1 and 2 are repeated to ensure the right parameters are being used.

1. **Choose parameters** – Select the parameters for the scheme, including modulus q, ring dimension n and error distribution ψ. These parameters should be the same as those used in the key generation process.
2. **Define the ring R_q** - Construct the polynomial quotient ring R_q = Z_q[X] / (Φ_n(X)), where Φ_n(X) = X^n + 1 is the n-th cyclotomic polynomial. This step ensures that the arithmetic operations during encapsulation are consistent with those used in key generation.
3. **Parse the receiver’s public key**- Obtain the receiver’s public key, which is a pair of polynomials (a,b) in R_q.
4. **Sample a random message ‘m’** - Choose a uniformly random message 'm' from the message space. The message space is a subset of R_q, and the choice of 'm' depends on the specific encoding used in the scheme.
5. **Encode the message ‘m’ -** Convert the message 'm' into a polynomial 'm_hat' in R_q using a suitable encoding function. This step ensures that the message is represented in the same mathematical structure as the other polynomials in the scheme.
6. **Sample a random polynomial ‘r’ –** Choose a random polynomial ‘r’ from the error distribution ψ. This introduces noise into the ciphertext to protect the ciphertext from eavesdroppers.
7. **Compute the ciphertext components ‘u’ and ‘v’ –** Calculate the two polynomials as follows:
   u = a * r
   v = b * r + m_hat
   These polynomials represent the encapsulated message and are designed so that only the receiver who has the secret key can recover the original message.
8. **Form the ciphertext ‘c’ –** Combine polynomials ‘u’ and ‘v’ to create the ciphertext ‘c’, which is a pair (u,v). This ciphertext can be securely transmitted to the receiver.
9. **Derive the shared secret ‘ss’ –** Compute the shared secret ‘ss’ by applying a hash function to the encoded message ‘m_hat’. Both the sender and receiver will be able to compute the same shared secret ‘ss’ which can be used to securely communicate.

7.12.3. Decapsulation

Decapsulation involves the receiver recovering the original message and the shared secret from the received ciphertext using their private key. This is done in the following steps.

1. **Choose parameters** - Select the parameters for the scheme, including the modulus q, the ring dimension n, and the error distribution ψ. These parameters should be the same as those used in the key generation and encapsulation processes.
2. **Define the ring R_q** - Select the parameters for the scheme, including the modulus q, the ring dimension n, and the error distribution ψ. These parameters should be the same as those used in the key generation and encapsulation processes.
3. **Parse the received ciphertext** - Obtain the ciphertext 'c' sent by the sender, which is a pair of polynomials (u, v) in R_q.
4. **Obtain the receiver's secret key** - Retrieve the receiver's secret key, which is a polynomial 's' in R_q.
5. **Compute the polynomial 'w'** - Calculate the polynomial 'w' in R_q as the difference between 'v' and the product of 'u' and 's', or w = v - u * s.
6. **Attempt to recover message 'm'** - Decode the polynomial 'w' to obtain an approximation of the original encoded message 'm_hat'. The decoding function used here should be the inverse of the encoding function used in the encapsulation process.
7. **Re-Encapsulate the message 'm'** - Using the recovered message 'm' and the receiver's public key (a, b), compute the ciphertext components 'u_prime' and 'v_prime':
   (a). u_prime = a * r
   v_prime = b * r + m_hat
   'r' is the polynomial used by the sender during encapsulation. The receiver does not know 'r' but due to the nature of the scheme, can still compute 'u_prime' and 'v_prime'.
8. **Verify the ciphertext** - Compare the received ciphertext (u, v) with the re-encapsulated ciphertext (u_prime, v_prime). If they are equal, the recovered message 'm' is considered valid.
9. **Derive the shared secret 'ss'** - Compute the shared secret 'ss' by applying the same hash function to the recovered message 'm_hat' as used in the encapsulation process.

7.12.4. Advantages of Kyber

1. **Quantum Resistance -** Kyber is designed to be secure against quantum attacks, making it a suitable replacement for current public key cryptosystems such as RSA and Elliptic Curves.
2. **Efficiency –** Kyber is computationally efficient regarding key generation and encapsulation/decapsulation operations. They involve polynomial arithmetic which can be performed quickly using **Fast Fourier Transform** or **Number Theoretic Transform**.
3. **Compact keys and ciphertext –** Kyber has small key sizes and ciphertexts relative to other post-quantum algorithms.
4. **Choice of parameters –** Kyber has parameter sets including Kyber512, Kyber768 and Jyber1024 which provide different trade-offs between security and performance allowing users to select which one suits them best.

5. **Simple structure –** Kyber is generally simple, with the mathematical concepts used easily to understand which aids in analysis.

7.12.5. Disadvantages of Kyber

1. **Exposure / life-span –** Lattice-based cryptography is still a new research topic that discoveries are constantly being made. This could lead to currently unknown vulnerabilities exposing the algorithm and rendering it useless.
2. **Larger keys and ciphertexts than classical algorithms –** One of the most notable trade-offs with quantum-resistant algorithms is the increased complexity yielding larger keys and ciphertexts, which in effect increase the storage and bandwidth needed for these.
3. **Error distribution –** Kyber makes use of error distributions such as discrete Gaussian or binomial distributions. Generating these efficiently and securely can be challenging, especially on weaker devices.

Kyber has many trade-offs to consider, is the increased complexity and security worth the extra resources and time needed for these algorithms? Despite these, Kyber is considered a promising candidate for post-quantum cryptography and was one of the finalists in the NIST post-quantum cryptography competition.

7.12.6. Key Terms

Ring Dimension – A ring dimension refers to the dimension of a ring as a vector space over its base field. A polynomial ring with coefficients in the field has a countably infinite dimension. A ring is equipped with two binary operations, addition, and multiplication. In Kyber, the ring dimension is related to which Kyber security level is chosen, which then determines the size of the polynomials used, A larger dimension leads to more complexity, larger keys and ciphertexts.

Error Distribution – Refers to the probability distribution of errors from the true value. It characterized randomness of errors. In terms of Kyber, it is used to select random polynomials to add noise to the Kyber scheme, increasing the hardness of the scheme.

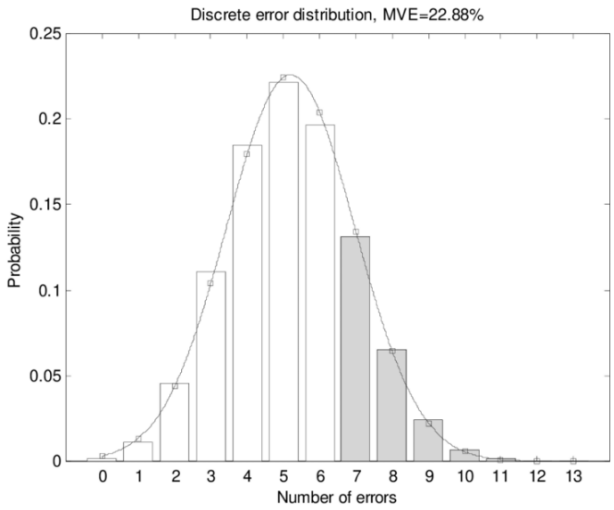


Figure 50: Error Distribution Example

Gaussian Distribution – Is a continuous probability distribution in a bell-shaped curve, defined by its mean (μ) and standard deviation (σ) It's widely used in analytics due to the Central Limit Theorem, which states that the sum of many independent, identically distributed random variables approaches a Gaussian distribution.

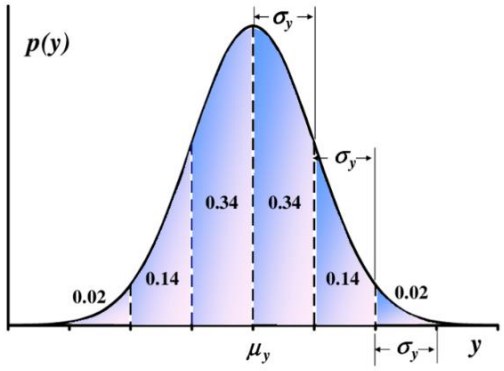


Figure 51: Gaussian Distribution Example

Binomial Distribution – This is a discrete probability distribution that models the number of successes in a fixed number of Bernoulli trials, each with the same probability of success. In Kyber, this can instead be used for the Gaussian distribution as it can offer a more efficient sampling.

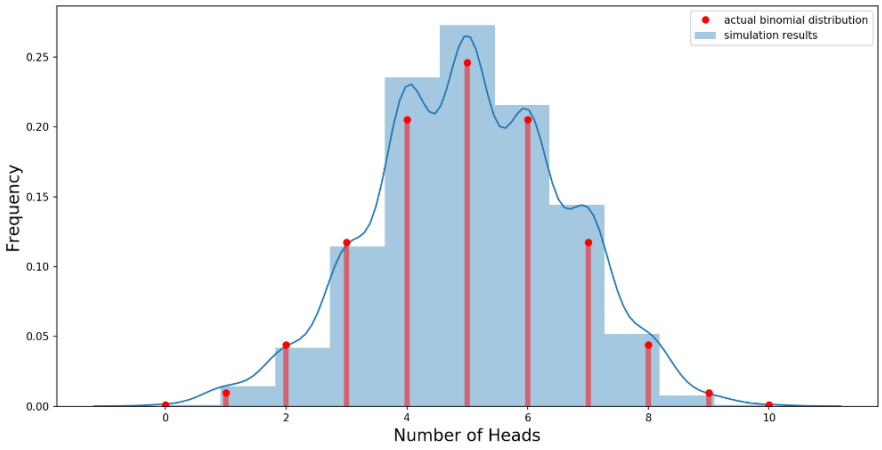


Figure 52: Binomial Distribution Example

Cyclotomic Polynomial (Φ_n(X)) – This is a special type of polynomial that has roots that are the nth primitive roots of unity. In Kyber, the nth cyclomatic polynomial, Φ_n(X) = X^n + 1, is used to define the polynomial ring R_q. Polynomials in this ring have coefficients modulo q.

Coefficients – Refers to the elements of the polynomials used in the scheme. Polynomials in Kyber have the following form, P(X) = c_0 + c_1 * X + c_2 * X^2 + ... + c_(n-1) * X^(n-1). In this, c_i are (for I – 0, 1… n-1) are the coefficients of the polynomial P(X). Each coefficient c_i is an integer that belongs to the set {0, 1… q-1}, it is an element in the ring Z_q. The coefficients of the polynomial influence the properties of the generated keys.

Uniformly Random – Refers to the property of selecting an element from a set such that each element has an equal probability of being chosen, this is done to ensure unpredictability and security.

Ideal Generated – Refers to the set of all polynomials that can be formed by multiplying the generating polynomial by any other polynomial. In Kyber, the idea generated by the nth cyclotomic polynomial is used to define the polynomial ring R-q.

Fast Fourier Transform (FFT) – An efficient algorithm for computing the **discrete Fourier transform (DFT)** and its inverse. FFT is used in Kyber to perform polynomial arithmetics efficiently.

Number Theoretic Transform (NTT) – A special case of DFT that operates in a ring of integers modulo a prime number. NTT is used to perform efficient polynomial arithmetic, as it allows fast multiplication of polynomials in ring R_q.

Discrete Fourier Transform (DTF) – Mathematical transformation to convert a signal or sequence of values into its frequency-domain representation. Given a sequence of complex numbers x_0, x_1… x_(N-1), DTF produces another sequence of complex numbers X_0, X_1…., X_(N-1) where X_k represents hr amplitude and phase of the k-th frequency component in the sequence. The DTF is defined as follows:

$$P(X) = c_0 + c_1 * X + c_2 * X^2 + ... + c_{(n-1)} * X^{(n-1)}$$

for k = 0, 1, … N-1 and I represents the imaginary unit (i^2 = -1).

This is used in lattice-based cryptography but is computationally expensive. NTT and DFT reduce the complexity of the operations to make this computationally faster.

7.13. CRYSTALS-Dilithium

In the following paragraphs, for any terms underlined and highlighted in **bold, see** the end of this section for definitions.
Like Kyber, Dilithium is based on lattice cryptography, specifically the Learning with Errors, Module-LWE and the Module Short Integer Solution problem. Despite this similarity, Dilithium instead is a quantum-resistant digital signature scheme (Anon., CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation).
Dilithium was selected by NIST in July 2022 as one of the four winners of their Post-Quantum Standardisation.

7.13.1. Key Generation

The main goal is to create a public key with a secret key that is mathematically retiled but computationally hard to link together without the secret key. This process has some similarities with Kyber but does differ as shown below.

1. ***Choose the security level*** – Dilithium offers three security levels, all with security and performance trade-offs: Dilithium-2, Dilithium-3, and Dilithium-5. These different levels will determine the specific parameters needed for the scheme such as the polynomial degree ‘n’, integer modulus ‘q’ and error distribution ‘η’.
2. ***Sample the secret and error polynomials*** – **(a)** Sample a secret polynomial ‘s’s from the error distribution ‘η’. This distribution is chosen to be a discrete Gaussian

distribution over the integers with a small **standard deviation**. The polynomial 's' will have an 'n' coefficient, where 'n' is the polynomial degree specified by the chosen security level.
(b) We must also sample an error polynomial 'e' from the error distribution η. The error polynomial also has 'n' coefficients.

3. ***Generate public polynomial 'a'*** – Sample a public polynomial 'a' uniformly at random from the ring R_q, where ring R_q is the ring of polynomials with coefficients in the range 0 to 'q-1' and 'q' is the integer modulus specified by the security level. The polynomial 'a' has 'n' coefficients.
4. ***Calculate the public polynomial 't'*** – Compute the public polynomial 't' as the product of t = a* s +r in R_q. The polynomial 't' has 'n' coefficients.
5. ***Output the key pair*** – **(a)** The public key is pair (a, t), this public key can be shared with anyone and is used for signature verification. **(b)** The secret key is polynomial 's'. This secret key must be kept private and is used for signing messages.

The security f the Dilithium scheme relies on the hardness of the Learning with Errors (LWE) and Ring-LWE problems. Given the only public key (a,t), it's believed to be computationally infeasible to recover the keypair or forge a signature without knowing the secret key.

7.13.2. Signing

The main goal of the signing process is to create a signature for a given message using a secret key so that the signature can be verified by anyone with the corresponding public key like follows.

1. **Compute the hash of the message** – Use cryptographic algorithms such as SHA256 or SHA-3 to generate a hash 'm' of the message you want to sign.
2. **Sample a random polynomial 'y'** – Sample a random polynomial 'y' from a distribution with bounded coefficients. The distribution used in Dilithium is typically a centred binomial distribution or a discrete Gaussian distribution with a small deviation. The polynomial 'y' has 'n' coefficients, where 'n' is the polynomial degree specified by the chosen security level.
3. **Compute w = a * y** – Calculate the product of public polynomial 'a' and the random polynomial 'y' in the ring R_q, where R_q is the ring of polynomials with coefficients in the range 0 – q-1, and 'q' is the integer modulus specified by the security level. The polynomial 'w' and 'n' coefficients.
4. **Generate a challenge** – Compute a hash c = H(m, w) using the message hash 'm' and the product 'w', where 'H' is the hash function used.
5. **Compute the response** – Calculate the response polynomial 'z' as the sum of the random polynomial 'y' and product of the challenge hash 'c' and secret polynomials; such that
   z = y + c * s. The polynomial 'z' has 'n' coefficients.
6. **Output signature** – The signature is the pair (z, c). The signature can be shared along with the message so that anyone with the corresponding public key can verify the message's authenticity.

The idea of the Dilithium signing process is to create a signature that can be verified efficiently using the public key, but also being computationally hard to forge without the secret key.

7.13.3. Verification

The main goal of verification is to check whether a given signature for a message is valid using the corresponding public key. If valid, it provides a guarantee that was message was signed by the holder of the secret key associated with the public key. The steps are as follows.

1. **Compute the hash of the message** – Generate a hash 'm' of the message, whilst making sure you are using the same hashing algorithm that was used by the signer.
2. **Compute 'w' from the signature** – Calculate 'w' so that w = a * z – c * t, using the signature components 'z' and 'c, and the public key components 'a' and 't', in the ring R_q.
3. **Generate a challenge –** Compute the hash ''c so that c = H(m,w) using the message hash 'm' and the product 'w', where 'H' is the hashing algorithm used in the signing steps.
4. **Check the signature –** Verify that the computed hash 'c' matches the hash 'c' from the signature received. If they match, the signature is valid, and you can be sure that the message was not altered in transit and that the sender of the message was indeed the person who signed said message.

Since hashes are one-way functions, we cannot just reverse the operations and work our way back. We need to recalculate the challenge hash 'c' based on the received message and signature and try to find a match to verify the message and its authenticity.

7.13.4. Advantages of Dilithium

1. **Post-Quantum Security** – Dilithium is resistant to both classical and quantum attacks, as even the Learning with Errors problem is computationally hard for a quantum computer to solve. Classic algorithms such as RSA or ECDSA are vulnerable to breaking broken via quantum computers, so Dilithium poses as a good candidate for digital signature in the future.
2. **Efficiency** – Like Kyber, Dilithium about other post-quantum algorithms has short signatures and public keys such as SPHINCS+. Its signing and verification stages are mathematically efficient making it suitable for a range of devices and applications.
3. **Random Oracles –** Dilithium does not rely on random oracles, which is an idealized model where hash functions have random oracles. This is seen as an advantage as schemes that rely on these are less secure.

7.13.5. Disadvantages of Dilithium

1. **Larger Key Sizes –** It is expected with the development of quantum-resistant algorithms that the key sizes will be much larger to provide more security. This does mean that it may not be as quick as the classic algorithms, especially on weaker devices.
2. **Life Span –** Like all post-quantum algorithms, Dilithium is a new scheme and hasn't had the optimisation that classical algorithms have received over the last few decades. This may lead to the discovery of unknown vulnerabilities which can harm the usage and development of this algorithm.
3. **No Forward Secrecy –** Most digital signatures, including Dilithium, do not initially implement forward secrecy. If the secret key is compromised, then a signature can be

forged. Forward secrecy can be implemented through other means but is not inherited in Dilithium itself.

7.13.6. Key Terms

Standard Deviation – This is a statistical measure that quantifies the amount of variation in a set of data values. It's used to understand how spread out the data points are from the average value. A small deviation indicates the points are clustered together whilst large means they are more spread out.
Regarding Dilithium, it's related to the error distribution used to sample secret and error polynomials, with them sampling from a discrete Gaussian distribution. This standard deviation determines the level of 'noise' introduced which affects security and efficiency. A smaller deviation leads to smaller coefficients, meaning smaller keys. However, a balance must be made with the deviation amounts as you don't want keys too small or big that will severely affect the security and performance of the algorithm.

Forward Secrecy – This s a property of cryptographic protocols that ensures the confidentiality of past communications, even if the long-term key is compromised. It essentially protects past sessions from being decrypted by an attacker. This is usually done using short new term keypairs for each session. As Dilithium does not inherently apply this, it means that if your secret key is compromised, an attacker can decrypt all past communications you made using that secret key.

Challenge – The challenge is a deterministic value based on the hash of the message and the value of 'w'. The challenge is used to bind the message to the signature and ensure it cannot be used for a different message or tampered with without it being detected. This makes it difficult for an attacker to forge a signature or modify it without the secret key.

7.14. PICNIC

In the following paragraphs, for any terms underlined and highlighted in **bold, see** the end of this section for definitions.
PICNIC is a post-quantum signature scheme based on the concept of zero-knowledge proofs and a non-interactive variant called non-interactive zero-knowledge proofs (NIZKPs) (Zaverucha, 2020). The primary goal is to allow a prover to convince a verifier that they know a secret without revealing any information about the secret itself.

7.14.1. Key Generation

PICNIC involves creating a private-public key pair. The private keypair is kept secret by the signer while the public key is shared with others who need to verify the signer's signature.

1. **Choose LowMC parameters** – Select the parameters for the LowMC block cipher, including the block size (n) and the key size (k)m the number of S-boxes (m), the number of rounds ® and the type of linear layer used. These parameters impact the security level and security of the scheme. Picnic has multiple levels of security – PICNIC-L1, PICNIC-L3 and PICNIC-L5. These all have two modes, PICNIC-FS which is optimized for speed over security, and PICNIC-FULL which focuses on security over speed.
2. **Generate a private key** – The private key (sk) is essentially the secret key for the LowMC block cipher. It is a randomly chosen k0bit string, where k is the key size

determined by the chosen LowMC parameters. The private key needs to be a **cryptographically secure** random number generator to generate the private key.

3. **Generate public key –** The public key (pk) in PICNIC consists of two parts; an input value (x) and the corresponding output value (y) of the LowMC block cipher when using the private key (sk) as the secret key.
   (a) Choose a random n-bit input value (x), where n is the block size determined by LowMC parameters.
   (b) Compute the output value (y) by encrypting the input value (x) using the LowMC block cipher with the secret key (sk).
   (c) The public key (pk) is the tuple (x,y). Tuple refers to an ordered / sequence of elements.
4. **Output the key pair –** The resulting key pair is (sk, pk), where sk is the private key (LowMC secret key) and pk is the public key (tuple (x,y). The signer keeps the private key secret and shares the public key with others to verify their signatures.

By using the LowMC block cipher, PICNIC can achieve its security goals whilst maintaining its efficiency in signing and verifying. The public (pk) serves as a commitment to the private key, and the zero-knowledge proof in the signature ensures the signer knows the corresponding private key without revealing it.

7.14.2. Signing

The signature scheme is built using the MPC-in-the-head approach, which turns a **multi-party computation** (MPC) protocol into a non-interactive zero-knowledge proof.

1. **Compute the message digest** – Hash the message, m, using a cryptographic hash function such as SHA256 or SHA3.
2. **Randomly partition the MPC protocol –** Choose a parameter t, representing the number of views in which the MPC will be partitioned.
   Generate t random seeds (s1 s2 …. St) and use them to create t random tapes (T1, T2, Tt) which serve as the source of randomness for the virtual parties in the MPC protocol.
3. **Run the MPC protocol for each view –** For each view Vi, execute the following steps:
   (a) Assign each of the ‘t’ parties a random tape ‘Ti’ and a share of the private key ‘ski’. The sum of all private key shares equals the actual private key (sk).
   (b) Emulate the MPC protocol for viewing Vi using random tapes and private key shares. The protocol computes the LowMC encryption with the secret key shares as input. The output of the MPC protocol for each view is an **intermediate value** (wi) and a transcript (Ti) containing the communication between all virtual parties.
   (c) Compute the commitment (Ci) for each view by hashing the initial state and the intermediate value (wi),
4. **Compute the challenge set –** The signer computes a challenge set by hashing the commitments of all views and the message digest. The challenge consists of c views, where c is a predetermined parameter. The views in this challenge set will be used to

construct the Zero-knowledge proof, whilst the other views will be used s responses to the challenge.

5. **Reveal non-challenged views -** The signer reveals the non-challenged views by providing the private key shares and the transcripts of these views. These revealed views will be used by the verifier to check the consistency of the proof.
6. **Generate the NIZKP –** The signer combined the non-challenged views and the commitments of the challenged views to construct the non-interactive zero-knowledge proof. This proof is the core component of the algorithm.
7. **Compute the signature –** The signer uses a hash function to compress the NIZKP and the message digest into a signature.

The resulting signature can be sent with the message to the recipient, who can verify its validity by issuing the public key and the verification process. The NIZKP in the signature proves the signer knows the private key corresponding to the public key without revealing its information.

7.14.3. Verification

The verification process checks the validity of the given signatures. The verifier does not need to know the signer's private key to perform this verification. The steps area is as follows.

1. **Compute the message digest** – Hash the message m, using the cryptographic hash function used in the signing process to obtain message digest d.
2. **Parse the signature** – Extract the NIZKP and the commitments of the challenged views from the signature. The NIZKP contains private key shares, random tapes, intermediate values, and transcripts for non-challenged views. It also contains the commitments for the challenged views and the message digest.
3. **Check the non-challenged views –** For each non-challenged view Vi, perform the following checks:
   (a) Verify that the private key shares when combined, result in the correct public key.
   (b) Verify that the random tapes Ti and the private key shares ski used in the MPC protocol for view Vi are consistent with the provided intermediate value wi and transcript Ti.
   (c) Verify that the commitment Ci of the non0challened view Vi matches the commitment included in the signature.
4. **Reconstruct the challenged views –** Using the transcripts Ti on the non-challenged views, the verifier can reconstruct the private key shares and random tapes of the challenged views. For each challenged view Vi, perform the following steps:
   (a) R4econstruct the private key shares ski and random tapes Ti from the transcripts of the non-challenged views.
   (b) Rerun the MPC protocol for view Vi using the reconstructed private key shares and random tapes to obtain the intermediate value wi.
   (c) Compute the commitment Ci by hashing the initial state and intermediate value wi.
5. **Verify the challenged views –** For each challenged view Vi, check the reconstructed commitment Ci matches the commitment included in the signature.
6. **Check the challenge set –** Compute the challenge hash h by concatenating the commitments of all views and the message digest and then hashing the result. Use the

challenge hash h to deterministically select the challenge set and check that it matches the challenge set used during the signing process.

7. **Determine the validity of the signature –** If all checks have passed, the signature is considered valid.

The verification process ensures the signature is only valid if it was created by the signer who knows the private key corresponding to the public key. It checks the consistency of all challenge views until the verifier is convinced of the signer's knowledge. This means that the verifier can technically repeat these steps over and over until they are convinced of the signer.

7.14.4. Advantages of PICNIC

Post-Quantum Security – PICNIC is believed to be secure against attacks from both classical and quantum computers until our current classical algorithms, meaning that it's an important candidate for the future of cryptography.

Efficiency – PICNIC uses the LowMC block cipher scheme with low multiplicative complexity, which is more efficient than other signature schemes in terms of computational power needed.

Side-Channel Resistance – Due to the use of MPCs, PICNIC has a higher resistance to side-channel attacks compared to other post-quantum schemes.

7.14.5. Disadvantages of PICNIC

Signature Size – PICNIC has relatively large signatures which leads to an increased need for communication and storage overhead.

Performance – Although PICNIC is computationally efficient, is it still relatively slow as there are a lot of steps involved in the scheme.

Choice of Parameters – PICNIC has a relatively large range of parameters to choose from, which can generally be hard to find a good balance of which parameters suit you the best.

7.14.6. Key Terms

LowMC Block Cipher – This is a customizable block cipher with low multiplicative complexity. It is efficient in terms of the number of AND gates used when implemented which is suitable for zero-knowledge proofs.

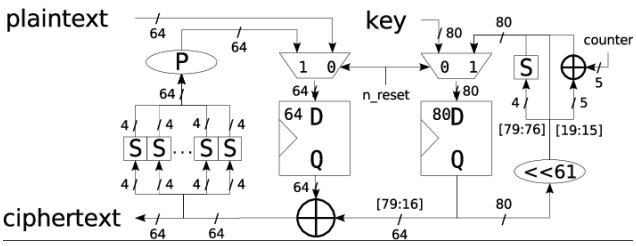


Figure 53: LowMC Block Cipher example

Multi-Party Computation – This is a subfield of cryptography that deals with the problem of securely computing a function with multiple inputs from different parties without revealing any of its information. This allows PICNIC to split up its private key shares to be used without revealing its contents.

Side-Channel Resistance – Refers to the ability of an algorithm to withstand side-channel attacks, which exploits information leaked through physical implementations such as timing information, power consumption or even electromagnetic radiation. The MPC process within PICNIC makes it hard for an attacker to correlate side-channel information to the secret key.

7.15. Falcon

In the following paragraphs, for any terms underlined and highlighted in **bold, see** the end of this section for definitions.
Falcon is a digital signature scheme based on the Fast-Fourier Lattice-Based Compact Signatures over NTRU) (Pierre-Alain Fouque, 2020). Falcon provides two levels of security Falcon-512 and Falcon-1024, with the numbers indicating the key sizes.
Falcon was selected by NIST in July 2022 as one of the four winners of their Post-Quantum Standardisation.

7.15.1. Key Generation

This process involves selecting appropriate parameters, choosing secret polynomials, and computing a key pair. The steps are as follows.

1. **Choose parameters** – **(a)**Select a degree ‘n’, in which ‘n’ determines the size of the polynomial ring Rq = Zq[X]/(X^n+1). Typically, ‘n’ is chosen as a power of two, and its value directly impacts the security level and performance of the algorithm. The larger ‘n’ is the more security it is providing but the more performance is impacted. **(b)** Select an integer ‘q’ from the modulus. ‘q’ should be a prime integer large enough to provide the desired security level. Like ‘n’, the larger this number is, the more it will impact security and performance.
2. **Generate secret polynomials** – Select two secret polynomials ‘f’ and ‘g’ with small coefficients. These coefficients should be chosen from a small range such as {-1,0,1}. These polynomials should be selected in a way that ensures they are **invertible** in the ring Rq. One method for generating these polynomials is to set the constant term of each polynomial to be odd, guaranteeing invertibility.
3. **Compute the private and public keys** – **(a)** Compute the public key ‘h’. This is done by multiplying the secret polynomial ‘g’ by the secret polynomial ‘f’ modulo ‘q’ so that h = g*f mod q. **(b)** Prepare the key pair. The private key consists of the secret polynomials’ ‘f’ and ‘g’ and the public key contains the polynomial ‘h’. These keys will be used for signing and verifying messages,

   It's important to maintain the secrecy of the private key while distributing the public key to recipients, otherwise, attacks can decrypt or even forge signatures.

7.15.2. Signing

The signing process with Falcon involves hashing the message, generating a **random nonce** computing the **syndrome,** and finding a short polynomial to create the signature. The signing steps are as follows.

1. **Hash the message** – Given a message m, compute its hash hm using a cryptographic hashing algorithm such as SHA245 or SHA3.

2. **Generate a random nonce** – Randomly generate a small polynomial ‘r’ with coefficients from a small range. This polynomial serves as a nonce value to ensure each signature is unique, even if the message is identical.
3. **Compute the syndrome** – Using the public key ‘h’ and the hash ‘hm’, compute the syndrome ‘s’/ The equation is s = r * hm mod q. This syndrome combines information about the message, nonce and the signer’s public key.
4. **Find a short polynomial** – Use an algorithm such as **Fast-Fourier Sampling (FFS)** to find a short polynomial ‘e’ such that e*f ≡ s mod q, where ‘f’ is the signer’s secret polynomial from the private key, ≡ denoting equivalence. The FFS is designed to find a short polynomial efficiently.
5. **Create the signature** – The signature of the message ‘m’ is the polynomial ‘e’ found in the previous step. This signature is sent to the recipient with the message.

The Falcon digital signature aims to provide strong security guarantees whilst maintaining efficiency.

7.15.3. Verification

The verification process is important to ensure the authenticity and integrity of a signed message. To verify a signature, you do the following.

1. **Hash the message -** Given the original message m, compute its hash ‘hm’ using the same hashing function used in the signing process.
2. **Compute the reconstructed syndrome –** Using the received signature ‘e’, public key ‘h’ and hash ‘hm’ of the message, compute the reconstructed syndrome ‘reconstructed’/ The equation is as follows: reconstructed = e * h – hm mod q. This step aims to reconstruct the syndrome compute during the signing.
3. Check signatures validity – Verify that the signature polynomial ‘e’ has small coefficients, indicating it’s a short polynomial. Falcon signatures must have small coefficients which are essential for their security. Compare the reconstructed syndrome ‘reconstructed’ with the original syndrome ‘s’ compute during the signing. If they match, the equation e*f ≡ s mod q is valid, and the signature is also valid.

If both the signature ‘e’ is a short polynomial and the reconstructed syndrome matches the original, the verification process confirms the signature is valid. This ensures the message’s authenticity and integrity, as only the private key holder could have generated a valid signature for that message.

7.15.4. Advantages of Falcon

Post-Quantum Security – Falcon is based on the hard lattice problems, which have been introduced into the post-quantum era. It is believed these hard lattice problems are both classic and quantum resistant.

Efficiency – Falcon is designed to be computationally efficient, allowing weaker systems to implement this sort of algorithm.

Small Key Sizes – Falcon offers smaller key sizes compared to other algorithms based on the Learning with Errors or code-based cryptography, leading to lower storage needs.

Strong Security Guarantees – Falcon's security is based on the Short Integer Solution and Learning with Errors which has been extensively studied and is hard for even quantum computers to solve.

7.15.5. Disadvantages of Falcon

Implementation – Implementing Falcon securely and efficiently can be challenging due to the complicated underlying maths and requirements for precise number operations. Implementing the Fast-Fourier Sampling can be difficult to do.

Side-Channel Attacks – If not implemented correctly, it can be left vulnerable to side-channel attacks. Extra defences will need to be implemented to further prevent these attacks, leading to decreased performance.

Larger Signature Sizes – Despite having relatively small key sizes, Falcon has larger signature sizes requiring more storage requirements to store.

If Falcon can be implemented correctly, it can be a very efficient and secure algorithm to use. Despite this, the implementation difficulty does have to be taken into consideration because if done incorrectly, it can lead to attacks.

7.15.6. Key Terms

Invertible – Refers to an element that has an inverse. If element 'a' is invertible, there exists another element 'b' such that the result of the operations between 'a' and 'b' yields the same element for that operation. If 'a'a has a modular inverse 'b' modulo 'm', then
(a * b) mod m = 1. This is important during the signing and verifying in Falcon as the secret polynomial 'f' must be invertible to ensure a unique short polynomial 'e' can be found satisfying the equation e * f ≡ s mod q.

Nonce – Short for 'number used once' is a random pseudo-random value used in cryptography to ensure the uniqueness and unpredictability of tasks. As mentioned earlier, the nonce is used to compute syndrome 's' to ensure each signature is unique even if two of the same messages are sent.

Syndrome – This is a value derived from a received message that contains information about the errors or alterations that may have occurred during transit. In Falcon, it is computed during the signing phase. During verification, the receiver reconstructs this syndrome, and it matches they have a security guarantee that the message was not altered during transit.

7.16. BIKE

In the following paragraphs, for any terms underlined and highlighted in **bold, see** the end of this section for definitions.
BIKE (Bit-Flipping Ken Encapsulation) is based on the learning with errors problem and uses a variant called Quasi-Cyclic Moderate Density Parity-Check (QC-MDPC) (co, 2021).

7.16.1. Key Generation

The key generation process in BIKE involves the creation of a public-private key pair. They are generated as follows.

1. **Generate two quasi-cyclic (QC) parity-check matrices** – Generate two binary quasi-cyclic parity-check matrices, H0 and H1, each with dimensions (n,r), where n is

the block length and r are the number of rows. These matrices have a fixed **Hamming weight.** This means that each row has t 1's and the remaining entries 0's. Quasi-cyclic matrices have a cyclic structure meaning each row is a circular shift of the previous row.

2. **Compute generator matrix G –** Compute the generator matrix G, which has dimensions (r,n,-r). G is the null space of the concatenated party-check matrix [H0 | H1]. G is a matrix that satisfies the equation [H0 | H1] * G = 0, where 0 is a zero matrix. You can calculate G using **Gaussian elimination**.
3. **Generate an error vector e of weight t –** Generate a binary error vector e of length n-r with a fixed Hamming weight t. The position of the ones in the error vector is chosen uniformly at random.
4. Calculate the public key (pk) – The public key is calculated by multiplying the generator matrix G by the error vector e so that pk = G * e. The result is the binary vector of length n-r.
5. Form the private key (sk) – The private key is composed of the two parity-check matrices H0 and H1, and the error vector e. It can be represented as sk = (H0, H1, e).

The key generation in BIKE involves generating two quasi-cyclic parity-check matrices, computing a generator matrix and an error vector, and calculating the private-public key pair. The public key is the product of the generator matrix and error vector, and the private key is composed of the parity0check matrices and error vector.

7.16.2. Encapsulation

The BIKE algorithm creates a shared secret and a ciphertext for the given public key. This process is typically executed by the sender of a message to ensure secure communication. The steps for encapsulation are as follows.

1. **Generate a random message vector m** – Generate a binary message vector m of length (n-r), where n is the block length and r is the number of rows in the parity check H0 and H1, The message vector will be used to create the shared secret and ciphertext.
2. **Compute u = m * pk** – Compute the vector u by multiplying the message vector m by public key pk. The public key has dimensions (n-r), and the resulting vector u will have the same dimensions.
3. **Generate an error vector e' of weight t** – Generate a binary error vector e' of length n-r with a fixed Hamming weight t. The position of the ones is chosen uniformly at random.
4. **Compute v = m * G + e' –** Compute vector v by multiplying the message vector m by the generator matrix G and adding the error vector e'/ The generator matrix G has dimensions (r, n, -r) and the resulting vector v will have dimensions (n-r).
5. **Create the ciphertext (ct) –** The ciphertext consists of the concatenated vectors u and v. The resulting ciphertext has dimensions (2*(n-r)). The ciphertext is transmitted to the recipient along with the public key to establish secure communication.
6. **Derive a shared secret (ss) using a key deviation function (KDF) –** Derive a shared secret, using a key derivation function (KDF) applied to the message vector m. The shared secret is used for further communication and encryption between the sender and the recipient.

The encapsulation step involves generating a random message vector, computing intermediate vectors u and v, creating a ciphertext by concatenating them and deriving a shared secret using a key derivation function applied to the message vector.

7.16.3. Decapsulation

The decapsulation process in BIKE is used by the recipient to recover the shared secret using the receiver ciphertext and their private key. This allows both parties to securely communicate and establish a shared secret key for further communication. The details are as follows.

1. **Parse the ciphertext (ct) into two vectors u and v** – Parse the ciphertext ct, into two separate vectors u and v, each of length (n0r), where n is the length of the block and r is the number of rows in the parity-check matrices H0 and H1.
2. **Compute the syndrome s = H0 * u + H1 * v** – Compute the syndrome vector s by multiplying the private keys parity-check matrix H0 by vector u, and the matH1 by vector v, and then add the results. The syndrome vector s has dimension (r).
3. **Use a decoding algorithm to find error vector e" close to the original error vector e –** Using an algorithm such as the Bit Flipping algorithm, to find the error vector e" that is close to the original vector e. The Bit Flipping Algorithm tries to correct errors in the received syndrome s by iteratively flipping bits in an initially zero-initialised error vector e". This is done by using the private key's parity-check matrices H0 and H1 until a criterion is met.
4. **Compute the message vector m' by subtracting e" from v –** Once you have obtained the error vector e", compute the recovered message vector m' by subtracting e" from the vector so that,' = v – e".
5. **Derive the shared secret ss, using the same key derivation function as in the encapsulation process –** Derive the shared secret using the same KDF as the encapsulation process and apply it to the recovered message vector m.

If the decapsulation process is successful, the derived shared secrets from both processes will be the same. This ensures communication between the sender and recipient is secure and allows them to establish a shared secret key for further communication.

7.16.4. Advantages of BIKE

Post-Quantum Security – BIKE is designed to be secure against classical and quantum attacks as it relies on the Learning with Errors problem, making it a strong candidate for the new era of cryptography.

Efficiency – Due to the usage of quasi-cyclic matrices and bit-flipping decoding, BIKE offers efficient key generation, encapsulation and decapsulation. This results in low computation complexity and reduced memory requirements on devices.

Hardware Implementation – BIKEs structure and bit operations allow high-speed and low-power devices to take advantage of the algorithm.

Simplicity – BIKE is relatively simple compared to other algorithms which make it easy to implement and analyse.

7.16.5. Disadvantages of BIKE

Large Key Sizes – Compared to classical algorithms, BIKE has large key sizes requiring more storage and transmission overhead.

Decoding Failure – The bit-flipping algorithm used by BIKE is generally good, it does have a really low risk of failure which can be of concern in certain applications.

Adaptability – Considering post-quantum algorithms are still new, a new attack can be discovered, which can question the defensive adaptability of algorithms such as BIKE.

7.16.6. Key Terms

Hamming Fixed Weight – Also known as a fixed weight, refers to the numbers of 1's in a certain binary vector and matrices. In BIKE, error vectors e and e' and H0, H1 have a fixed Hamming weight of t, meaning they contain t amount of 1's.

Gaussian Elimination – This is a linear algebra technique used to solve systems of linear equations by reducing a given matrix to a row echelon. In BIKE, this is used to compute the generator matrix G from the concatenated parity-check matrix [H0 | H1].

7.17. Rainbow

In the following paragraphs, for any terms underlined and highlighted in **bold, see** the end of this section for definitions.
The Rainbow post-quantum is a multivariate cryptographic signature scheme. It relies on multivariate polynomial equation problems to be quantum resistant (Jintai Ding, 2019).

7.17.1. Key Generation

The key generation process includes choosing parameters, generating central and affine maps, and then generating the key pair.

1. **Choose parameters –** Select the parameters of the scheme including:
   (a) The number of variables (n), which is the total number of variables in the system.
   (b) The number of questions (m), which is the number of equations in the system.
   (c) The number of layers (u). The rainbow scheme is a generalisation of the **Unbalanced Oil and Vinegar** (OUV) scheme. The parameter u represents the number of layers in the structure, with u – 1 corresponding to the original OUV scheme. Each layer has a different number of OUV variables to increase efficiency and security.
2. **Generate the central map F –** Randomly generate the central polynomial map F, which consists of m quadratic polynomials in n variables. Each polynomial can be represented as a sum of **monomials** over a finite field, with randomly chosen coefficients. To ensure that F is an invertible function, it should be constructed so that its **Jacobian matrix** has full rank. This means that the linearization of the polynomials at every point in the finite field should be linearly independent.
3. **Generate affine maps S and T -** Randomly generate two invertible affine maps S and T. Both S and T are linear maps followed by a translation. Specifically, S: x ↦ S1x + s2 and T: y ↦ T1y + t2, where x and y are input and output vectors, S1 and T1 are invertible matrices, and s2 and t2 are translation vectors. The dimensions of S1 and T1 are nxn and mxm, respectively, while the dimensions of s2 and t2 are nx1 and mx1.

4. **Public and Private key pair -** The public key is the composition of the three maps: P = T(F(S(x))), where x is the input. The private key consists of the central map F and the two affine maps S and T. To obtain the public key, you first apply the affine map S to the input x, then apply the central map F, and finally apply the affine map T.

The key generation process involves selecting appropriate parameters, generating a central polynomial map F, and creating two affine maps S and T. The public key is the composition of these maps whilst the private key consists of the individual maps.

7.17.2. Signing

The signing process as usual starts with hashing the message first. We then run into a unique method provided by this algorithm as we then invert the public key and maps to get the output signature. The steps are explained as follows.

1. **Hash the message –** Compute the hash of the message you want to sign using a chosen hash function. This function should map the message to an m-dimensional vector over a finite field. This hashed message, denoted as h, serves as a commitment to the original message.
2. **Invert the public key -** Using the private key, find the **preimage** of the hash under the public key. This step involves inverting the maps T, F, and S in reverse order. The result will be the signature of the message and are as follows:
   (a) Compute the inverse of the affine map T to obtain the intermediate value y, such that y = T^(-1)(h). This involves finding the inverse of the matrix T1 and subtracting the translation vector T2 -> y = T1^(-1)(h - t2).
   (b) Compute the inverse of the central map F to obtain the intermediate value x', such that x' = F^(-1)(y). This step is computationally demanding as it solves a system of m quadratic equations in n variables.
   (c) Compute the inverse of the affine map S to obtain the signature x such that x = S^(-1)(x'). This involves finding the inverse of the matrix S1 and subtracting the translation vector s2: x = S1^(-1)(x' - s2).
3. **Output the signature –** The resulting vector x is the signature of the message. This signature can be transmitted alongside the original message to provide authentication.

The signing process in the Rainbow signature scheme involves hashing the message to obtain a fixed-length commitment, inverting the public key using the private key, and outputting the resulting signature.

7.17.3. Verification

Since we used the inverse of the maps and keys to compute the signature, we now must apply the maps and keys normally. Which is unorthodox from the usual methods.

1. **Hash the message –** Compute the hash of the message using the same hash function used during the signing process. This function should map the message to an m-dimensional vector over a finite field.
2. **Apply the public key –** Apply the public key P to the signature x and compute P(x). The public key is the composition of three maps: P = T(F(S(x))) so to apply the public key we do the following:
   (a) Apply the affine map S by computing S(x) by multiplying the matrix S1 with the

signature vector x and adding the translation vector s2: S(X) = S1x + s2.
(b) Apply the central map F by computing F(S(x)) and evaluating the m quadratic polynomials that form the central map F with the input S(x). Each polynomial is evaluated over a finite field.
(c) Apply the affine maps Y by computing T(F(S(x))) by evaluating the m quadratic polynomials that form the central map F with the input S(x). Each polynomial is evaluated over a finite field.

3. **Check equality -** Verify if the output of the public key applied to the signature is equal to the hash of the message: P(x) == h. If they are equal, then the signature is valid, and the message is authentic. Otherwise, this would mean either the signature was tampered with, or the incorrect private key was used to verify the signature.

The verification step is unique in how it almost seems to work backwards from how you would usually sign signatures, but this does inherit good security implications of the algorithm.

7.17.4. Advantages of Rainbow

Post-Quantum Security – Rainbow relies on the multivariate polynomial equations problem, which is believed to be resistant to classical and quantum attacks.

Efficiency – Rainbow offers very fast signature verification with its use of polynomial evaluations and simple linear transformation when generating the keys.

Usage of the UOV scheme – Rainbow is a generalisation of the UOV scheme, which leads to optimisation and increased security. It also adds flexibility with parameter selections.

7.17.5. Disadvantages of Rainbow

Large Key Sizes – The key sizes in Rainbow are relatively large compared to not only classic algorithms but also other post-quantum signatures. This is due to the multiple quadratic polynomials with many coefficients.

Expensive Signing Process – Despite having a quick verification process, Rainbow has a very expensive signing process due to the need to invert the central map which requires solutions to multiple multivariate quadratic equations. The parameters used could also make this even more expensive to compute.

7.17.6. Key Terms

Monomials - A monomial is a mathematical expression that consists of a single term formed by the product of one or more variables, each raised to a non-negative integer power. Within Rainbow, monomials are the building blocks of the quadratic polynomials that constitute the central map F.

Jacobian matrix: The Jacobian matrix is a matrix of all first-order partial derivatives of a vector-valued function. In Rainbow, the Jacobian matrix plays a crucial role in ensuring that the central map F is invertible.

$$f(x_1, x_2, \ldots, x_n) = (f_1, f_2, \ldots, f_m)$$

$$J_f = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

Figure 54: Jacobian Matrix

Central Maps - Refers to a set of m quadratic polynomials in n variables over a finite field. The central map is a crucial part of the private key and serves as the core component of the scheme.

Affine Maps - A linear transformation followed by a translation. In Rainbow, two affine maps, S and T, are used to disguise the central map F, forming the public key. Both S and T are part of the private key, and their purpose is to add an extra layer of complexity to the system.

7.18. Sphincs+

In the following paragraphs, for any terms underlined and highlighted in **bold, see** the end of this section for definitions.
Sphincs+ is a **stateless** signature scheme that is also hash-based meaning it relies on hash functions for its security (Jean-Philippe Aumasson, 2022). Sphincs+ was the only hash-based signature scheme selected as the NIST finalist in 2022.

7.18.1. Key Generation

Since Sphincs+ relies on hash functions, the methods are somewhat more traditional and closer to the classical algorithms. It is to be remembered that hash functions aren't like the other classifications of other classical methods as they are still somewhat quantum resistant as they do not rely on the mathematical issues mentioned earlier. Whilst the smaller hash sizes will be under threat, hash functions such as SHA-256 are believed to still be viable, but steps have been taken to develop SHA-3 which is a more secure hashing algorithm.

1. **Generate a secret seed –** The secret seed is a randomly generated bit string that generates secret key elements. It should have enough entropy to ensure security. The length of the secret seed depends on the hash function itself. You have 128-bit security using SHA-256, the decrease in bit security is due to the birthday attack.
2. **Generate a public seed -** The public seed is another random bit string that is part of the public key. It is used to generate public key elements such as addresses and random values for the **Merkle tree**.

3. **Initialize Address (AD) -** The address is a structure used to identify nodes in the trees and is essential for the construction and traversal of the Merkle tree. It includes several fields, such as layer, tree, and node positions. The address is not secret and can be shared with others.
4. **Generate the private key –** Once these seeds are generated using a cryptographically secure random number generator, you then initialize the address structure with appropriate values for the fields.
5. **Generate leaf nodes -** The leaf nodes of the Merkle tree contain the public keys of the **WOTS+** or **FORS** instances, which are derived from the secret keys. Using the secret seed and address structure, generate the secret keys for each WOTS+ and FORS instance. Then, compute the corresponding public keys for each instance using the public seed and addresses.
6. **Hash the leaf nodes -** Use a cryptographic hash function to hash the public keys of the WOTS+ and FORS instances. These hashed values serve as the leaf nodes of the Merkle tree.
7. **Compute intermediate nodes -** Starting from the leaf nodes, compute the intermediate nodes of the Merkle tree by hashing the children's nodes. The intermediate nodes' values are computed as the hash of the concatenation of their children's nodes' values.
8. **Compute the root -** Continue computing the intermediate nodes until you reach the top level of the tree. The single node at the top level is the root of the Merkle tree, which serves as the public key for the SPHINCS+ scheme.

Within Sphincs+, the Merkle tree plays a large role in constructing the keypairs and their elements.

7.18.2. Signing

Rainbow differs from other signature schemes as it generates WOTS+ signatures and uses a FORS tree to create the signatures. The steps are as follows.

1. **Hash the message –** Hash the message to obtain a message digest.
2. **Create a randomizer –** Generate a random value, known as a randomizer, which is used to randomize the signature.
3. **Generate a Winternitz One-Time Signature (WOTS+) Signature –** First choose an unused WOTS+ instance from the Merkle tree to sign the message digest. Then use the WOTS+ private key corresponding to the chosen instance to sign the digest.
4. **Create a Forest of Random Subsets (FORS) tree -** Choose an unused FORS instance from the Merkle tree to sign the WOTS+ public key. Use the FORS private key corresponding to the chosen instance to sign the WOT+ public key.
5. **Compute Authentication Paths –** For every WOTS+ signature and FORS tree, compute the authentication paths. These paths are needed to verify the signatures and consist of sibling nodes in the Merkle tree that led from the signed leaves up to the root.
6. **Create the signature –** Combine the randomizer, WOTS+ signatures, FORS trees, and authentication paths to create the final SPHINCS+ signature.

The signing process in SPHINCS+ combines the message digest, a randomizer, WOTS+ signatures, FORS trees, and authentication paths to generate a non-deterministic, quantum-

resistant signature. SPHINCS+ brings a lot of uniqueness and a variety of options, as some of the other signatures tend to work in similar fashions. SPHINCS+ reused secure classical methods and integrates them with quantum-resistant methods to create an interesting signature scheme.

7.18.3. Verification

Verifying in SPHINCS+ will require us to reconstruct authentication paths and verify the WOTS+ signature and FORS tree. The steps are as follows.

1. **Hash the message –** Hash the message with the same algorithm used in the signing phase to obtain a message digest.
2. **Recover Public Keys -** Use the received WOTS+ signature and the message digest to recover the WOTS+ public key. This involves iteratively applying the hash function to the signature and the message digest according to the **Winternitz** parameter (w) until the original public key is recovered.
   Use the received FORS tree signature and the recovered WOTS+ public key to recover the FORS public key. This involves iteratively applying the hash function to the signature and the WOTS+ public key according to the FORS parameters (t, k, a) until the original FORS public key is recovered.
3. **Compute the Merkle Tree Root –** We must first recover the WOTS+ and FORS public keys and the received authentication paths to reconstruct the Merkle Tress internal nodes. Iteratively hash the recovered public keys with their siblings from the paths to move up the tree until you reach the top-level nodes. Once here, hash them together to compute the root of the Merkle Tree.
4. **Verify the Merkle Root –** Compare the Merkle Tree root with the public key provided by the sender. If they match the signature is valid and the message is authentic. If not, the message should not be trusted.

Recovering the public keys from WOTS+ and FORS almost feels like a game, that is how unique this algorithm looks. SPHINCS+ uses hash functions, which are already researched extensively meaning that the algorithm has a high level of security.

7.18.4. Advantages of SPHINCS+

Post-Quantum Security – Since SPHINCS+ relies on hash functions, it is resistant to quantum attacks so long as they keep using up-to-date algorithms such as SHA3. SHA-256 is viable for now but might be worth using large SHA key sizes to mitigate future security risks.

Stateless – This means it does not require a signer to maintain state information between signatures, making it practical to use.

High-Security Level – As mentioned, it has a high level of security, due to the parameters and use of hash functions, WORS+ and FORS.

Simplicity – SPHINCS relies on well-studied security properties that have been previously implemented in classical cryptography.

7.18.5. Disadvantages of SPHINCS+

Large Signature Size – SPHINCS+ has a large signature size compared to classical algorithms.

Complexity - SPHINCS+ requires more computations for signing and verification than some classical signature schemes.

Parameters - While the ability to tune parameters in SPHINCS+ can be seen as an advantage, it can also complicate the selection process and make it challenging to find the right balance between security, performance, and signature size.

7.18.6. Key Terms

Entropy - Refers to the degree of randomness or uncertainty in a set of data. High entropy is crucial for generating secure cryptographic keys. It makes keys hard to guess.

Merkle Tree - A Merkle tree is a tree data structure where each non-leaf node is the hash of its children nodes' values, and each leaf node represents a hash of some data. The root of the Merkle tree is a unique value that represents the combined hashes of all the data in the tree.

In SPHINCS+, the root of the Merkle tree is the public key of the SPHINCS+ scheme.

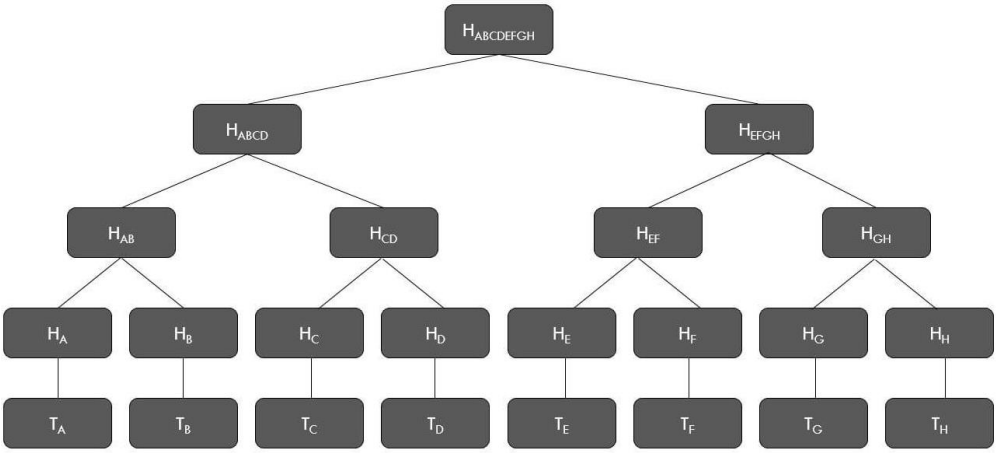


Figure 55: Merkle Tree Example

8. Conclusion

It's no doubt that at this time, the future of cryptography will be reliant on Lattice-based problems, especially after three of the four NIST competition winners being Lattice-based algorithms in CRYSTALS-Kyber, CRYSTALS-Dilithium and Falcon. We can see with the Ring-LWE and Module-LWE variants that optimisations and improvement are happening at a fast rate.
Despite the influence of Lattice-based cryptography, it was refreshing to see SPHINCS+ gain the last finalist spot, being a more classical algorithm than the others relying on hash functions and Merkle Trees with WOTS+ and FORS. There has also seen to be a massive increase of digital signature algorithms in the post-quantum space, with all the finalists except Dilithium being digital signature based. These digital signatures show the relevance of hash functions, as they are still a main component in generating signatures even in post-quantum algorithms.

It is to be assumed that post-quantum algorithms will be harder to run, and take up more space, but is that worth taking a small hit or the increased security it provides in the long run? Are the performance drops as big as expected, or can a modern computer today run them without any issue? That's the aim of the benchmarks, to determine whether we would benefit from moving to post-quantum cryptography worldwide as soon as possible.

9. Bibliography

Anon., CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation. *Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsk,* s.l.: s.n.

Chi, D., 2015. *Lattice-Based Cryptography for Beginners,* s.l.: DP Chi.

co, N. A. a., 2021. *BIKE Bit Flipping Key Encapsulation,* s.l.: s.n.

Emspak, J., 2022. *Jesse Emspak.* [Online]
Available at: https://www.space.com/31933-quantum-entanglement-action-at-a-distance.html
[Accessed 25 January 2023].

Ethereum, 2022. *What are zero-knowledge proofs?,* s.l.: Ethererum.

Geeksforgeeks, 2021. *RSA and Digital Signatures.* [Online]
Available at: https://www.geeksforgeeks.org/rsa-and-digital-signatures/
[Accessed 27 December 2022].

Geeksforgeeks, 2022. *Non-Interactive Zero Knowledge Proof,* s.l.: Geeksforgeeks.

Huang, Y., n.d. *Public-Key Cryptography from new Multivariate Quadratic Assumptions,* s.l.: International Association for Cryptologic Research.

Java T Point, n.d. *RSA Encryption Algorithm.* [Online]
Available at: https://www.javatpoint.com/rsa-encryption-algorithm
[Accessed 27 December 2022].

Jean-Philippe Aumasson, D. J. B. W. B., 2022. *SPHINCS+,* s.l.: s.n.

Jintai Ding, M.-S. C. A. P. D. S., 2019. *Rainbow,* Santa Barbara: s.n.

Jon, 2021. *SHA-3 Explained in Plain English.* [Online]
Available at: https://chemejon.wordpress.com/2021/12/06/sha-3-explained-in-plain-english/
[Accessed 14 January 2023].

Joyce, D., 2016. *What is the meaning of irreducible in algebra?* s.l.:s.n.

Khan, S., n.d. *What is the Twofish encryption algorithm?.* [Online]
Available at: https://www.educative.io/answers/what-is-the-twofish-encryption-algorithm
[Accessed 20 January 2023].

Levasseur, A. D. &. K., 2021. *Polynomial Rings.* [Online]
Available at:
https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics/Applied_Discrete_Structures_(Doerr_and_Levasseur)/16%3A_An_Introduction_to_Rings_and_Fields/16.03%3A_Polynomial_Rings
[Accessed 2 March 2023].

Martin, D., 2022. *https://sha256algorithm.com/.* [Online]
Available at: https://sha256algorithm.com/
[Accessed 27 December 2022].

Mathsisfun, n.d. *Polynomials.* [Online]
Available at: https://www.mathsisfun.com/algebra/polynomials.html
[Accessed 2 March 2023].

Metwalli, S. A., 2023. *What Is Superposition?.* [Online]
Available at: https://builtin.com/software-engineering-perspectives/superposition
[Accessed 25 January 2023].

Mustafeez, A. Z., 2015. *What is CTR?.* [Online]
Available at: https://www.educative.io/answers/what-is-ctr
[Accessed 27 December 2022].

NIST, 2022. *Post-Quantum Cryptography* , s.l.: NIST.

OBE, P. B. B., 2018. *Everything You Wanted To Know about Integer Factorization, but Were Afraid To Ask.* [Online]
Available at: https://medium.com/coinmonks/integer-factorization-defining-the-limits-of-rsa-cracking-71fc0675bc0e
[Accessed 27 12 2022].

Pierre-Alain Fouque, J. H. ,. K., 2020. *Falcon: Fast-Fourier Lattice-based,* s.l.: s.n.

Rakhade, K., 2020. *Shor's Algorithm (for Dummies).* [Online]
Available at: https://kaustubhrakhade.medium.com/shors-factoring-algorithm-94a0796a13b1
[Accessed 30 December 2022].

Roberto Avanzi, J. B. L. D. E. K. T. L., 2021. *CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation,* s.l.: s.n.

Schneier, B., 2020. *RSA-250 Factored.* [Online]
Available at: https://www.schneier.com/blog/archives/2020/04/rsa-250_factore.html
[Accessed 27 December 2022].

Sidhpurwala, H., 2023. *A Brief History of Cryptography.* [Online]
Available at: https://www.redhat.com/en/blog/brief-history-cryptography
[Accessed 27 December 2022].

Singh, H., 2019. *Code based Cryptography:,* Delhi: Scientific Analysis Group.

University of Basel, 2020. *Adding noise for completely secure communication.* [Online]
Available at: https://www.sciencedaily.com/releases/2020/06/200611133116.htm
[Accessed 11 March 2023].

Vacca, J., 2014. Advanced Data Encryption. In: J. R. Vacca, ed. *Cyber Security and IT Infrastructure Protection.* s.l.:Elsevier Inc, pp. 325-345.

Zaverucha, G., 2020. *The Picnic Signature Algorithm,* s.l.: s.n.