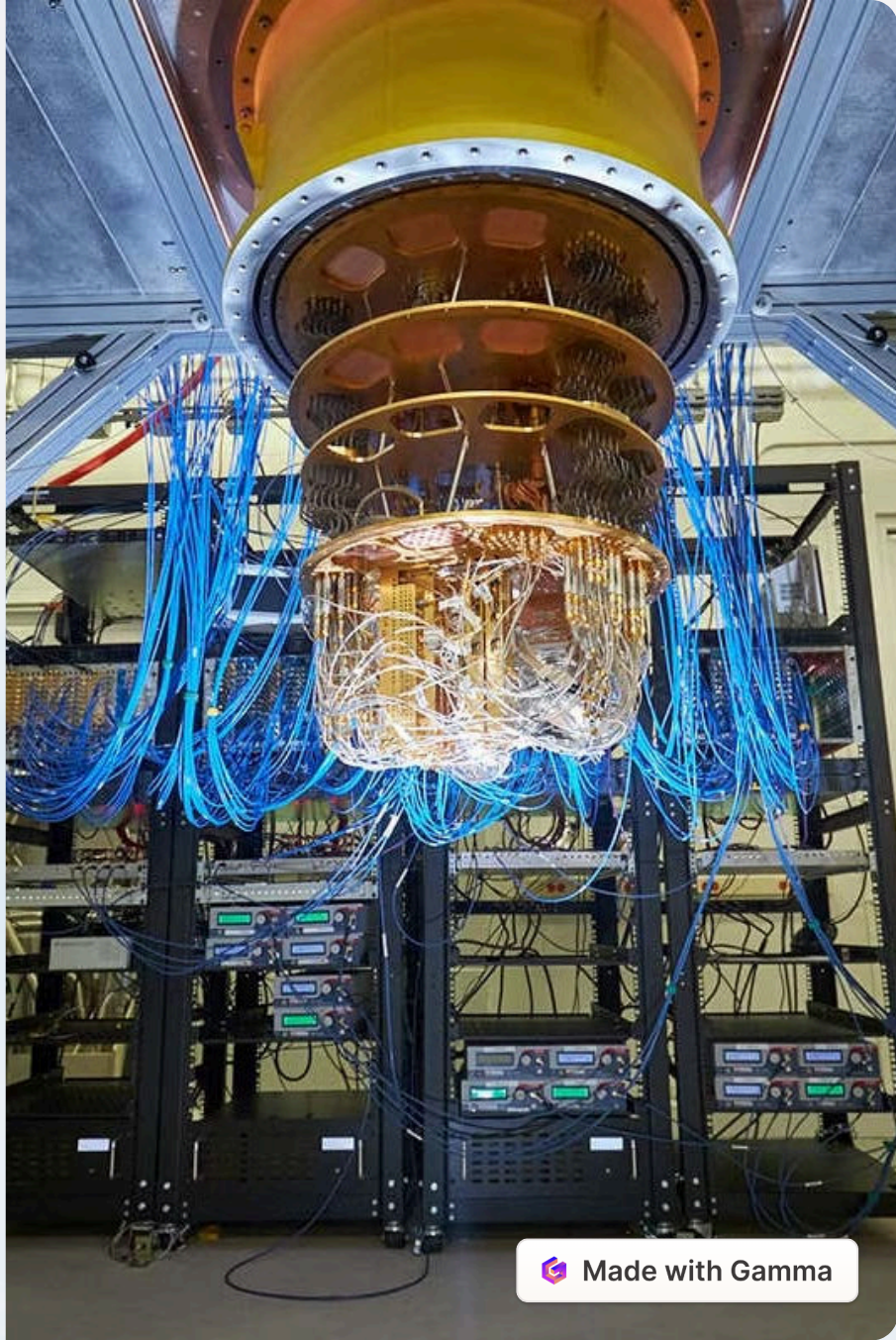
From Classical to Quantum: Parallelizing AES, RSA Algorithm on CPU, GPU, and Shor's Algorithm on Quantum Computers

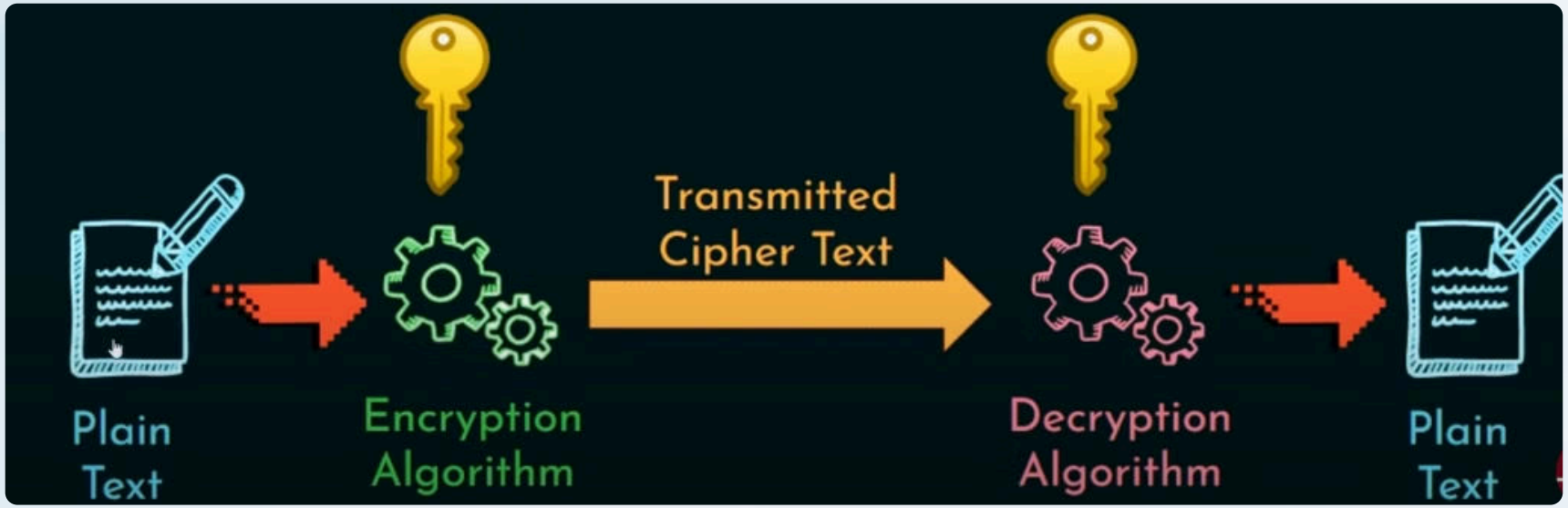


Introduction

Quantum computing has the potential to revolutionize various fields, particularly cryptography. Traditional cryptographic algorithms, such as AES and RSA, rely on computational hardness assumptions which can be threatened by quantum algorithms like Shor's algorithm. This project aims to understand and implement these algorithms, comparing their performance on classical (CPU and GPU) and quantum backends

Introduction to Cryptography

Cryptography is the art and science of securing communication and information in the digital age. It involves various techniques and algorithms to ensure the confidentiality, integrity, and authenticity of data.



Cryptography Fundamentals

Encryption

The process of transforming readable data (plaintext) into an unreadable format (ciphertext) to protect it from unauthorized access.

Decryption

The reverse process of converting ciphertext back into its original plaintext form, making the information readable again.

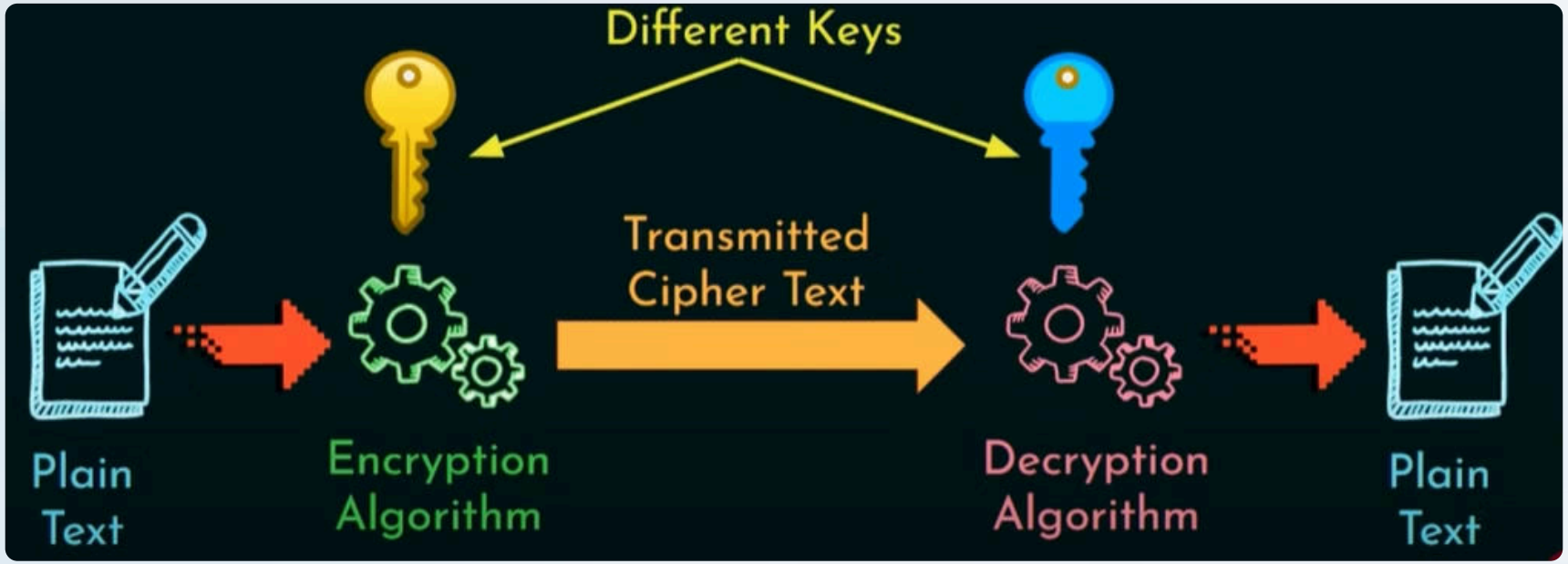
Key Management

The secure generation, distribution, and storage of cryptographic keys, which are essential for both encryption and decryption.

Types of Cryptography

Symmetric key cryptography: It is a type of encryption scheme in which the similar key is used both to encrypt and decrypt messages.

Asymmetric key cryptography: It is a type of encryption also known as public key cryptography is a method of cryptography that uses two different keys to encrypt and decrypt data.



Types of Cryptographic Algorithms

Block Ciphers

Encrypt and decrypt data in fixed-size blocks, such as the Advanced Encryption Standard (AES).

Public-Key Algorithms

Use a public-private key pair for encryption and digital signatures, like RSA.

Quantum Computing

Quantum computing is a revolutionary paradigm in the field of computation that leverages the principles of quantum mechanics to perform calculations that are infeasible for classical computers. Unlike classical computing, which relies on bits as the basic unit of information, quantum computing uses quantum bits, or qubits, which possess unique properties that enable unprecedented computational power.

Qubits

Qubits are the fundamental units of information in quantum computing, analogous to classical bits in traditional computing. However, qubits differ significantly in their behavior and capabilities due to their quantum mechanical nature:

- **Superposition**: Qubits can exist in multiple states (0 and 1) simultaneously, allowing quantum computers to process a vast amount of information in parallel.
- **Entanglement**: Qubits can be entangled, meaning the state of one qubit is directly related to the state of another, no matter the distance between them. This property enables complex correlations that classical computers can't achieve.
- **Quantum Gates**: Operations on qubits are performed using quantum gates, which manipulate qubit states through unitary transformations. Common gates include the Hadamard gate, Pauli-X gate, and CNOT gate.
- **Shor's Algorithm** is a quantum algorithm designed to efficiently factorize large integers. This poses a significant threat to RSA encryption, as it can break the encryption by finding the prime factors of large numbers in polynomial time.

What is qiskit?

Qiskit is an open-source quantum computing software development framework developed by IBM. It allows users to create, manipulate, and run quantum circuits on quantum computers and simulators. Here are some key aspects of Qiskit:

1. **Quantum Circuit Composition**: Qiskit provides tools for composing quantum circuits using its Python-based programming interface. Users can define quantum circuits by specifying quantum gates and operations applied to qubits.
2. **Quantum Simulators**: Qiskit includes quantum simulators that allow users to simulate the behavior of quantum circuits on classical computers. These simulators are useful for testing and debugging quantum algorithms before running them on actual quantum hardware.
3. **Quantum Hardware Access**: Qiskit provides access to IBM's quantum hardware through the IBM Quantum Experience platform. Users can run their quantum circuits on real quantum processors, accessing them via the cloud.
4. **Quantum Algorithms**: Qiskit includes implementations of various quantum algorithms, such as Grover's search algorithm, Shor's factoring algorithm, and algorithms for quantum machine learning and optimization.
5. **Quantum Circuit Visualization**: Qiskit offers tools for visualizing quantum circuits, allowing users to visualize the structure of their circuits and understand their behavior.

What are States?

In quantum computing, states represent the possible conditions or configurations of qubits, the fundamental units of quantum information. These states include basic states like |0⟩ and |1⟩, as well as more complex states like superposition states and entangled states. Understanding and manipulating these states is essential for performing quantum computations. By leveraging the unique properties of quantum states, such as superposition and entanglement, quantum algorithms can potentially solve certain problems much faster than classical algorithms. Thus, the purpose of states in quantum computing is to encode and process information in ways that exploit quantum phenomena to enable powerful computational capabilities.

(a) A 4x1 universal random quantum circuit

Gates

The purpose of gates in quantum computing is to manipulate the quantum states of qubits to perform specific quantum operations. These operations enable the implementation of quantum algorithms and the execution of quantum computations. Gates are the building blocks of quantum circuits, where they control the flow of quantum information and facilitate the execution of quantum algorithms. By applying gates to qubits in various combinations, quantum computers can perform complex calculations, simulations, and optimizations that are beyond the capabilities of classical computers. In summary, gates play a crucial role in encoding, processing, and manipulating quantum information, enabling the power and potential of quantum computing.

(b) The “bristle-brush” pattern formed by the gates applied to

Basic Quantum Gate

Hadamard Gate (H):

- Creates superposition by putting a qubit into an equal-weighted superposition of |0⟩ and |1⟩ states.

The Hadamard gate is represented by the following matrix:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Let's denote the basis states as $|0\rangle$ and $|1\rangle$, representing the computational basis states of a qubit.

When the Hadamard gate H is applied to the $|0\rangle$ state:

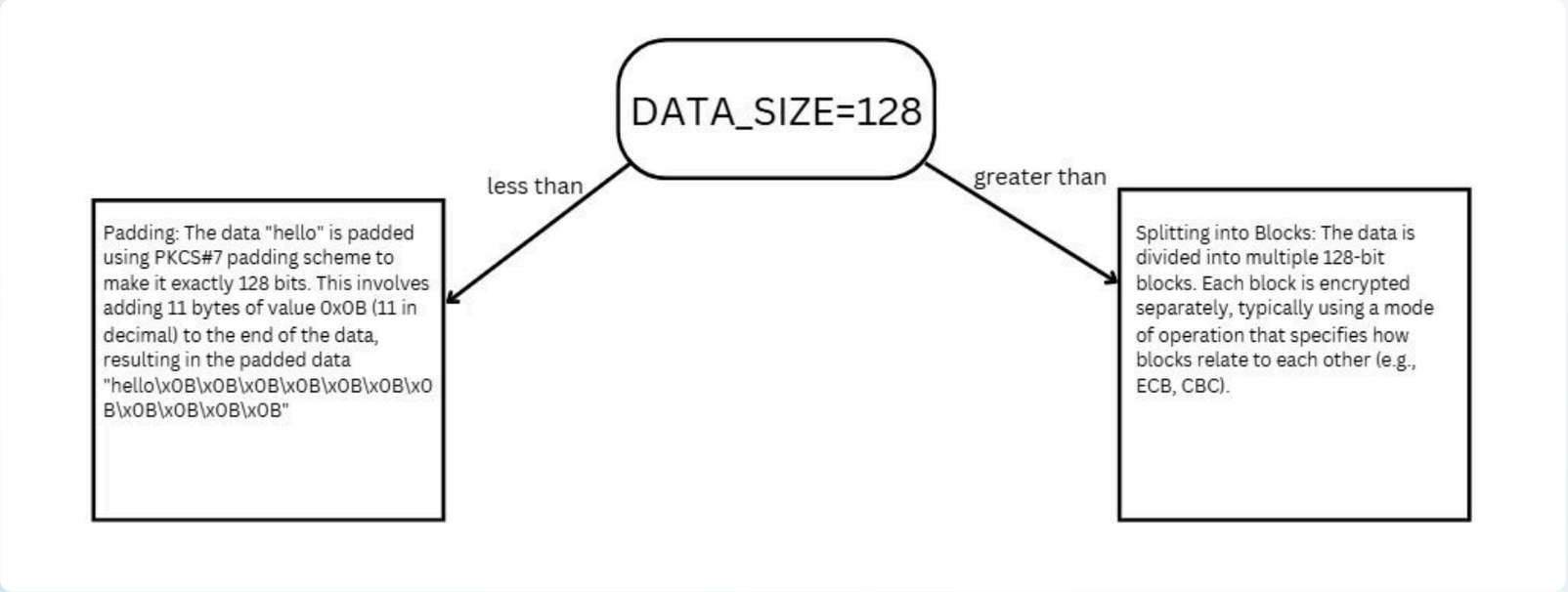
$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \cdot 1 + 1 \cdot 0 \\ 1 \cdot 1 + (-1) \cdot 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$$

Similarly, when the Hadamard gate H is applied to the $|1\rangle$ state:

$$H|1\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \cdot 0 + 1 \cdot 1 \\ 1 \cdot 0 + (-1) \cdot 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

So, applying the Hadamard gate H to the $|0\rangle$ and $|1\rangle$ states creates superposition states, with equal probability amplitudes.

AES (Advanced Encryption Standard) Algorithm



No. of rounds	Key size (in bits)
10	128
12	192
14	256

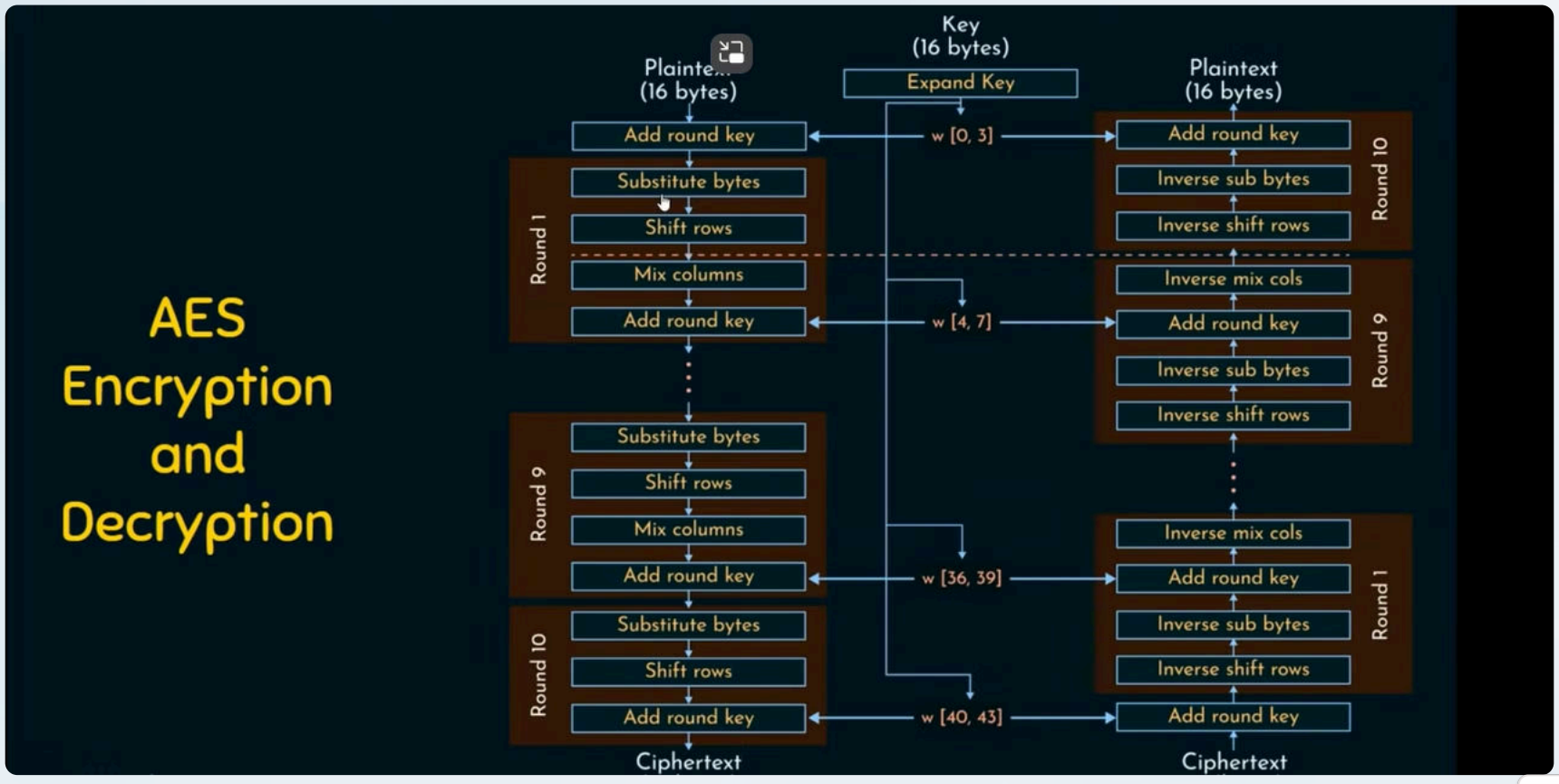
Encryption Process

The AES encryption process consists of the following steps:

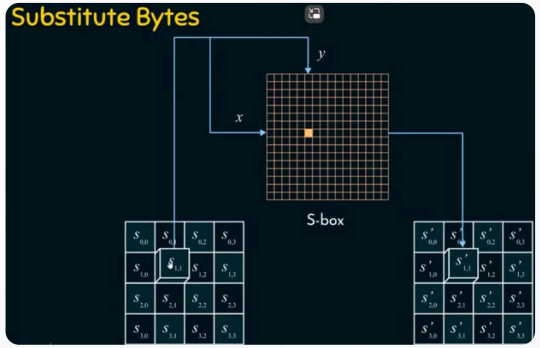
1. The plaintext is divided into blocks
2. **SubBytes:** implements the byte substitution
3. **Shift rows:** Each row is shifted a specific number of times
4. **Mix columns:** matrix multiplication is performed, where each column is multiplied with a matrix
5. Add round keys

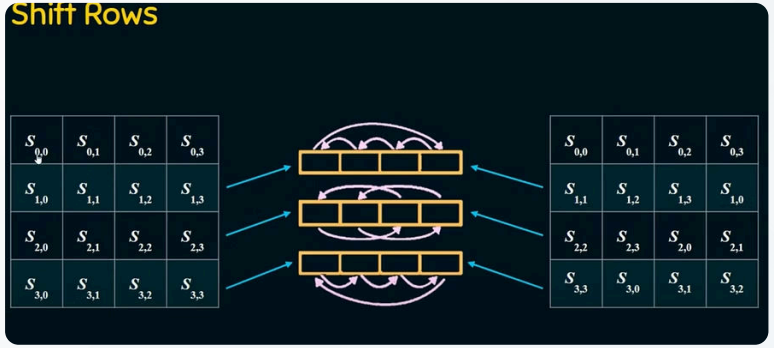
The process is repeated multiple times, where the number of rounds corresponds to the key length. For example, a 128-bit key requires 10 rounds, while a 256-bit key requires 14 rounds.

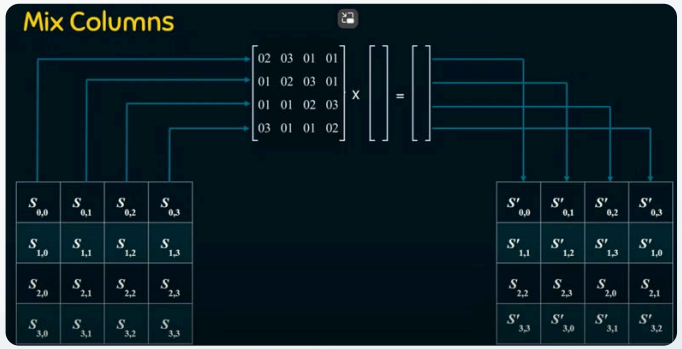
Once the final round is complete, the final ciphertext is produced.

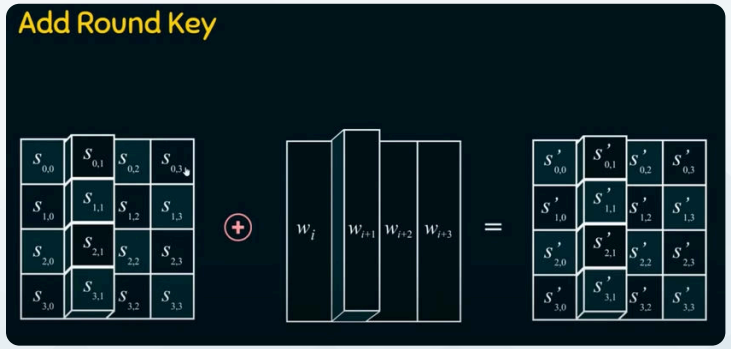


Round Transformations









AES Example - Input (128 bit key and message)

Key in English: Thats my Kung Fu (16 ASCII characters, 1 byte each)

Translation into Hex:

T	h	a	t	s		m	y		K	u	n	g		F	u
54	68	61	74	73	20	6D	79	20	4B	75	6E	67	20	46	75

Key in Hex (128 bits): 54 68 61 74 73 20 6D 79 20 4B 75 6E 67 20 46 75

Plaintext in English: Two One Nine Two (16 ASCII characters, 1 byte each) Translation into Hex:

T	w	o		O	n	e		N	i	n	e		T	w	o
54	77	6F	20	4F	6E	65	20	4E	69	6E	65	20	54	77	6F

Plaintext in Hex (128 bits): 54 77 6F 20 4F 6E 65 20 4E 69 6E 65 20 54 77 6F

AES Example - The first Roundkey

- Key in Hex (128 bits): 54 68 61 74 73 20 6D 79 20 4B 75 6E 67 20 46 75
- $w[0] = (54, 68, 61, 74)$,

$w[1] = (73, 20, 6D, 79)$,

$w[2] = (20, 4B, 75, 6E)$,

$w[3] = (67, 20, 46, 75)$

- $g(w[3])$:
- circular byte left shift of $w[3]$: $(20, 46, 75, 67)$
- Byte Substitution (S-Box): $(B7, 5A, 9D, 85)$
- Adding round constant (01, 00, 00, 00) gives: $g(w[3]) = (B6, 5A, 9D, 85)$
- $w[4] = w[0] \oplus g(w[3]) = (E2, 32, FC, F1)$:

0101 0100	0110 1000	0110 0001	0111 0100
1011 0110	0101 1010	1001 1101	1000 0101
1110 0010	0011 0010	1111 1100	1111 0001
E2	32	FC	F1

- $w[5] = w[4] \oplus w[1] = (91, 12, 91, 88)$,

$w[6] = w[5] \oplus w[2] = (B1, 59, E4, E6)$,

$w[7] = w[6] \oplus w[3] = (D6, 79, A2, 93)$

- first roundkey: E2 32 FC F1 91 12 91 88 B1 59 E4 E6 D6 79 A2 93

AES Example - Add Round key, Round 0

•State Matrix and Round key No.0 Matrix:

[54 4F 4E 20] [54 73 20 67]

[77 6E 69 54] [68 20 4*B* 20]

[6F 65 6E 77] [61 6*D* 75 46]

[20 20 65 6F] [74 79 6*E* 75]

•XOR the corresponding entries, e.g., 69 ⊕ 4*B* = 22

0110 1001

0100 1011

0010 0010

•the new State Matrix is

[00 3C 6E 47]

[1F 4E 22 74]

[0*E* 08 1*B* 31]

[54 59 0*B* 1*A]*

AES Example - Round 1, Substitution Bytes

•current State Matrix is

[00 3C 6E 47]

[1F 4E 22 74]

[0*E* 08 1*B* 31]

[54 59 0*B* 1*A]*

•substitute each entry (byte) of current state matrix by corresponding entry in AES S-Box

•for instance: byte 6E is substituted by entry of S-Box in row 6 and column E, i.e., by 9F

•this leads to new State Matrix

[63 *EB* 9*F* *A*0]

*[C*0 2*F* 93 92]

[AB 30 *AF* *C*7]

[20 *CB* 2*B* *A*2]

•this non-linear layer is for resistance to differential and linear cryptanalysis attacks

AES Example - Round 1, Shift Row

•the current State Matrix is

[63 *EB* 9*F* *A*0]

*[C*0 2*F* 93 92]

[AB 30 *AF* *C*7]

[20 *CB* 2*B* *A*2]

•four rows are shifted cyclically to the left by offsets of 0,1,2, and 3

•the new State Matrix is

[63 *EB* 9*F* *A*0]

[2F 93 92 C0]

[AF C7 AB 30]

[A2 20 CB 2B]

•this linear mixing step causes diffusion of the bits over multiple rounds

AES Example - Round 1, Mix Column

•Mix Column multiplies fixed matrix against current State Matrix:

[02 03 01 01] [63 EB 9F A0] [BA 84 E8 18]

[01 02 03 01] [2F 93 92 C0] [75 A4 8D 40]

[01 01 02 03] [AF C7 AB 30] [F4 8D 06 7D]

[03 01 01 02] [A2 20 CB 2B] [7A 32 0E 5D]

• entry *BA* is result of $(02 \cdot 63) \oplus (03 \cdot 2F) \oplus (01 \cdot AF) \oplus (01 \cdot A2)$:

• $02 \cdot 63 = 00000010 \cdot 01100011 = 11000110$

• $03 \cdot 2F = (02 \cdot 2F) \oplus 2F = (00000010 \cdot 00101111) \oplus 00101111 = 01110001$

• $01 \cdot AF = AF = 10101111$ and $01 \cdot A2 = A2 = 10100010$

•hence

11000110

01110001

10101111

10100010

10111010

AES Example - Add Round key, Round 1

•State Matrix and Round key No.1 Matrix:

[*BA* 84 *E*8 1*B*] [*E*2 91 *B*1 *D*6]

[75 *A*4 8*D* 40] [32 12 59 79]

[*F*4 8*D* 06 7*D*] [*FC* 91 *E*4 *A*2]

[7*A* 32 0*E* 5*D*] [*F*1 88 *E*6 93]

•XOR yields new State Matrix

[58 15 59 CD]

[47 B6 D4 39]

[08 1C E2 DF]

[8*B* *BA* *E*8 *CE*]

•AES output after Round 1: 58 47 08 8B 15 B6 1C BA 59 D4 E2 E8 CD 39 DF CE

Similarly round 2 to round 9 are done.

AES Example - Round 10

•after Substitute Byte and after Shift Rows:

[01 3A 8C 21] [01 3A 8C 21]

[33 3E B0 E2] [3E B0 E2 33]

[3D B8 8E 04] [8E 04 3D B8]

[BC 4D 1C A7] [A7 BC 4D 1C]

•after Round key (Attention: no Mix columns in last round):

[29 57 40 1A]

[C3 14 22 02]

[50 20 99 D7]

[5F F6 B3 3A]

•ciphertext: 29 C3 50 5F 57 14 20 F6 40 22 99 B3 1A 02 D7 3A

Decryption Process

1.Add Round Key : Each byte of the state array is combined with a byte of the round key using bitwise XOR (exclusive OR) operation.

2.Inverse Shift Rows : Inverse Shift Rows performs the inverse cyclical rotation of Shift Rows (you shift right instead of left).

3.Inverse Sub Bytes : Inverse Sub Bytes applies the inverse S-box to each byte of the state.

4.Inverse Mix Columns : Used to undo the mixing of columns that occurred during encryption. It involves matrix multiplication with a specific fixed matrix. The fixed matrix used in the inverse Mix Columns operation is the inverse of the matrix used in the Mix Columns operation during encryption. This operation provides diffusion and confusion to ensure the security of the cipher.

RSA (Rivest-Shamir-Adleman) Algorithm

Key Generation	RSA uses a public-private key pair, where the public key is used for encryption and the private key is used for decryption.
Encryption	Data is encrypted using the recipient's public key, ensuring only the intended recipient can decrypt the message.
Decryption	The encrypted data is decrypted using the recipient's private key, which is kept secret.
Digital Signatures	RSA also enables the creation of digital signatures to verify the authenticity and integrity of data.

RSA Algorithm

The RSA algorithm is a cornerstone of public-key cryptography, a system that uses two mathematically linked keys for encryption and decryption. It allows for secure communication by keeping one key private and the other publicly available.

Key Generation:

1. **Prime Numbers:** The foundation of RSA lies in two very large prime numbers, `p` and `q`. Choosing large primes is crucial for security as factoring a large number into its primes becomes computationally expensive.
2. **Modulus (n):** We calculate the modulus, `n`, by multiplying these prime numbers: `n = p * q`. This value is used in both the public and private keys.

Public key

- Accessible to everyone.
- Used for encryption (making data unreadable).

1. **Public Exponent (e):** We select a small integer, e, that is relatively prime to (p-1)(q-1). This means e has no common factors with (p-1)(q-1) other than 1.
2. **Public Key:** The public key is a pair consisting of the modulus n and the public exponent e: (n, e). This key is freely distributed to anyone who wants to send you encrypted messages.

Private Key

- Kept secret by the intended recipient.
- Used for decryption (recovering the original data).

1. **Private Key Derivation:** The private key, which must be kept secret, is derived from p, q, and e. We use a mathematical concept called the Extended Euclidean Algorithm to find another integer, d, such that: $de \equiv 1 \pmod{(p-1)(q-1)}$ This basically means that when d and e are multiplied together and the result is divided by (p-1)(q-1), the remainder is 1.

Encryption Process

1. **Plaintext Conversion:** The message to be encrypted (plaintext) is converted into a numerical format suitable for encryption. Padding schemes like PKCS#1 v1.5 might be used for larger messages.
2. **Encryption Function:** The message `m` is encrypted using the public key `(n, e)` with the following function: `E(m) = m^e (mod n)`. Here, the message `m` is raised to the power of the public exponent `e` and the result is taken modulo the modulus `n`. This mathematical operation transforms the message into an unreadable form called ciphertext `c`.

Decryption Process

1. **Decryption Function:** Only someone with the private key d can decrypt the ciphertext. They use the following function: D(c) = c^d (mod n). Here, the ciphertext c is raised to the power of the private exponent d and the result is again taken modulo the modulus n.
2. **Original Message Recovered:** This mathematical operation reverses the encryption process, recovering the original message m.

Security Considerations

- **Key Size:** The security of RSA relies heavily on the size of the prime numbers used. Larger key sizes provide stronger security but also require more computational power.
- **Key Management:** Proper key generation, storage, and distribution practices are crucial. Leakage of the private key compromises the entire system.
- **Potential Vulnerabilities:** While mathematically secure, RSA can be susceptible to side-channel attacks that exploit implementation weaknesses to extract information.

Simple Example

RSA Algorithm Example

Using the RSA algorithm in the example below, for p=5 and q=17, m=7 first encrypt and then decrypt.

- Choose p = 3 and q = 11
- Compute n = p * q = 3 * 11 = 33
- Compute φ(n) = (p - 1) * (q - 1) = 2 * 10 = 20
- Choose e such that 1 < e < φ(n) and e and φ(n) are coprime. Let e = 7
- Compute a value for d such that (d * e) mod φ(n) = 1. One solution is d = 3 // [(3 * 7) mod 20 = 1]
- Public key is (e, n) => (7, 33)
- Private key is (d, n) => (3, 33)
- The encryption of m is $c = m^e \bmod n$. The encryption of $m = 2$ is $c = 2^7 \bmod 33 = 29$.
- The decryption of c is $m = c^d \bmod n$. The decryption of $c = 29$ is $m = 29^3 \bmod 33 = 2$.

example with string "HELLO"

1. **Key Generation**:
   - p=11,q=13
   - $n=p\times q=11\times 13=143$
   - $\phi(n)=(p-1)\times(q-1)=10\times 12=120$
   - Public key: $(e,n)=(7,143)$
   - Private key: $(d,n)=(103,143)$

1. **Encryption**:
    - Message: "HELLO"
    - ASCII values: 'H' = 72, 'E' = 69, 'L' = 76, 'L' = 76, 'O' = 79
    - Encrypt each ASCII value using the public key:
        - 'H' encrypted: $72^7 \bmod 143 = 43$
        - 'E' encrypted: $69^7 \bmod 143 = 97$
        - 'L' encrypted: $76^7 \bmod 143 = 34$
        - 'L' encrypted: $76^7 \bmod 143 = 34$
        - 'O' encrypted: $79^7 \bmod 143 = 47$
    - Encrypted message: "43 97 34 34 47"

1. **Decryption**:
    - Decrypt each encrypted value using the private key:
        - '43' decrypted: 43^103mod 143=72
        - '97' decrypted: 97^103mod 143=69
        - '34' decrypted: 34^103mod 143=76
        - '34' decrypted: 34^103mod 143=76
        - '47' decrypted: 47^103mod 143=79
        - The decrypted message is "HELLO".

RSA in real world

To handle messages longer than the modulus 'n' in RSA encryption, we would typically use a technique called "**RSA encryption scheme with blocks**" or "**RSA encryption scheme with a secure padding scheme**" (like OAEP or PKCS#1 v1.5). These schemes involve breaking the message into blocks, padding the blocks to ensure they are the correct size, encrypting each block separately, and then combining the encrypted blocks into the final encrypted message.

Shor's Algorithm

- Let N is the number for which we want to find the prime factors
- Choose a random integer a such that 1<a<N
- Calculate GCD(a,N)
- If GCD(a,N)≠1, this GCD is a non-trivial factor of N
- The goal is to find the order r of a modulo N, which is the smallest integer r such that

$$a^r \equiv 1 \pmod{N}.$$

- **Quantum Subroutine**:
  - Prepare a superposition of all possible values.
  - Use Quantum Fourier Transform (QFT) to determine the period r.
- If r is even, compute

$$a^{r/2} \pm 1.$$

- Use the quantum subroutine again to find a new order 'r' for the new 'a'. Continue this until an even 'r' is found.
- Compute the GCD of N with

$$a^{r/2} - 1 \text{ and } a^{r/2} + 1.$$

- . These values are potential factors of N.
- If the factors found are trivial (i.e., 1 or N), repeat the process with a different random a.

Shors Algorithm Example

Shor's Algorithm Example: $N = 15$, so $p = 3, q = 5$

Step 1: Choose and a such that $gcd(a, 15) = 1$, for this example we choose $a = 7$

Step 2: Use a quantum computer to estimate r. We need r such that $7^r \equiv 1\ mod(15)$. Using the algorithm we find $r = 4$

Step 3: Calculate

$$gcd(7^{4/2} - 1, 15) = gcd(48, 15) = 3$$

$$gcd(7^{4/2} + 1, 15) = gcd(50, 15) = 5$$

Execution

Execution of Shor's algorithms using quantum backend

```
ATTEMPT 1:
Register Reading: 00000000
Corresponding Phase: 0.0
Result: r = 1

ATTEMPT 2:
Register Reading: 00000000
Corresponding Phase: 0.0
Result: r = 1

ATTEMPT 3:
Register Reading: 11000000
Corresponding Phase: 0.75
Result: r = 4
Guessed Factors: 3 and 5
*** Non-trivial factor found: 3 ***
*** Non-trivial factor found: 5 ***
```

Conclusion

This project focused on the implementation and comparative analysis of the AES and RSA algorithms on CPU and GPU platforms, and the execution of Shor's algorithm on a quantum backend. Here are the key conclusions drawn from our work:

AES and RSA on CPU and GPU

CPU Performance:

- Traditional implementations of AES and RSA on CPUs utilized well-established libraries like Crypto.Cipher for AES and Crypto.PublicKey for RSA.
- CPUs provided acceptable performance but were limited by their sequential processing capabilities.

GPU Performance:

- Implementing AES and RSA on GPUs significantly improved execution times due to the parallel processing capabilities of GPUs.
- We observed considerable speedups in encryption and decryption times for AES with different key sizes (128-bit, 192-bit, and 256-bit). Specifically, encryption and decryption times were reduced, demonstrating the advantage of leveraging GPU power for cryptographic tasks.

Shor's Algorithm on Quantum Backend

- **Quantum Implementation:** Using IBM's Qiskit, Shor's algorithm was successfully implemented on a quantum backend. Results indicated that even for small integers, the quantum computer could factorize numbers efficiently, showcasing the potential of quantum computing in solving problems currently infeasible for classical computers.

Comparision Graphs of AES and RSA