Implementation and Analysis of Classical Cipher Mechanisms in C++

Muhammad Ismail 2023452
Mahad Aqeel 2023286
Afeef Bari 2023356
Shayan 2023656

Information Security CY-211

Abstract

The following work focuses on analyzing the feasibility of using different kinds of classical cipher algorithms in C++. The primary focus is on three cipher types which include Symmetric, Asymmetric and Hybrid. Symmetric includes Caesar Cipher, Rail Fence Cipher, Playfair Cipher, Vigenère Cipher, OTP whereas asymmetric includes RSA encryption. The working of each cipher is described with regards to both encryption and decryption; the principles of mathematics underlying each cipher are also discussed on this basis of code snippets.

A general outline of the advantages and disadvantages of each cipher's algorithm is discussed and this is followed by the actual implementation of these ciphers using C++ programs. The project also discusses these ciphers in relation to the complexity and the time taken to encrypt and decrypt a message.

Furthermore, the program enables users to type plaintext and choose cipher to get the corresponding cipher text. The findings are expected to improve the knowledge of the casual reader on cryptographic concepts as well as the embedding of an actual code in C++. From these concepts students will be able to understand the current issues and developments taking place in information security.

TABLE OF CONTENT

1. Project Introduction

The modern world is seeing itself involved in issues of information security more and more often. A major component of data security, cryptography is the science of protecting communications via the use of code. This project offers a sound and informative Cryptography System which can show diverse encryption methods, so that people have a better understanding of the basics of data protection.

How the Project Will Work

The project on hand for evaluation is a menu-based simple encryption system that is user input based where they can select and apply different types of encryption to the input text. The current model provides a user-friendly interface where the user may test various forms of encryption for different cryptography purposes.

Functional Description

Upon launching the system, users are presented with a main menu to select the type of encryption they wish to perform:

- **Symmetric Encryption:** Has a single key – encryption and decryption both are performed quickly and effectively yet the key has to be shared safely.
- **Asymmetric Encryption:** Uses a combination of a public and a private key improving the security of systems used in communication as well as the authentication phase but it is slow due to its complexity.
- **Hybrid Encryption:** An amalgamation of the two: E.g AES protects the transfer of a session key used for acceleration of data encryption. This approach is efficient enough, and at the same time provoking a very high level of security, and that is why this approach is suggested to be applied to secure communication systems.

After selecting a category, users can choose from a range of specific encryption methods, including but not limited to:

- Symmetric Methods: There is Caesar Cipher, Vigenère Cipher, Rail Fence Cipher, Playfair Cipher, One-Time Pad.
- Asymmetric Methods: RSA Encryption.
- Hybrid Methods: A mix of techniques under one category, symmetric and one in the asymmetric category.

2. Advantages and Disadvantages of the Cipher Mechanisms

Caesar Cipher

Advantages:

- Simple to implement and understand.
- Minimal computational power required.

- Useful for educational purposes to demonstrate basic encryption.

Disadvantages:

- Extremely insecure; can be broken easily using frequency analysis.
- Fixed shift provides minimal complexity.
- Not suitable for modern applications.

Vigenère Cipher

Advantages:

- More secure than Caesar Cipher due to polyalphabetic substitution.
- Resistant to simple frequency analysis.
- Easy to implement with manageable keys.

Disadvantages:

- Vulnerable to key-repetition attacks (e.g., Kasiski examination).
- Key management can be challenging for longer texts.
- Not suitable for large-scale secure communication.

Rail Fence Cipher

Advantages:

- Easy to implement and understand.
- Provides obfuscation through transposition.
- Effective for simple scenarios with low-security needs.

Disadvantages:

- Vulnerable to brute-force attacks if the number of rows is small.
- Easily broken with known plaintext-ciphertext pairs.
- Lacks key strength for modern cryptographic needs.

Playfair Cipher

Advantages:

- Encrypts digraphs (pairs of letters), making it more secure than monoalphabetic ciphers.
- Resistant to frequency analysis of individual letters.
- Simple to use with a predefined key matrix.

Disadvantages:

- Vulnerable to frequency analysis of digraphs.
- Limited security compared to modern encryption methods.
- Requires a pre-shared key for both parties.

One-Time Pad

Advantages:

- It is provably secure only if the key is indeed random and used only once.
- Provides perfect secrecy.
- Impossible to crack without access to the key.

Disadvantages:

- Must be used with a key as long as the message, the distribution of the key becomes complicated.
- Vulnerable if the key is reused.
- Impractical for frequent or large-scale communication.

RSA Encryption

Advantages:

- Provides strong security through large key sizes.
- Eliminates the need for pre-shared keys using public/private key pairs.
- Widely used in secure communications and digital signatures.

Disadvantages:

- Computationally expensive compared to symmetric encryption.
- Key generation and management are complex.
- Vulnerable to quantum computing attacks in the future.

Hybrid Method

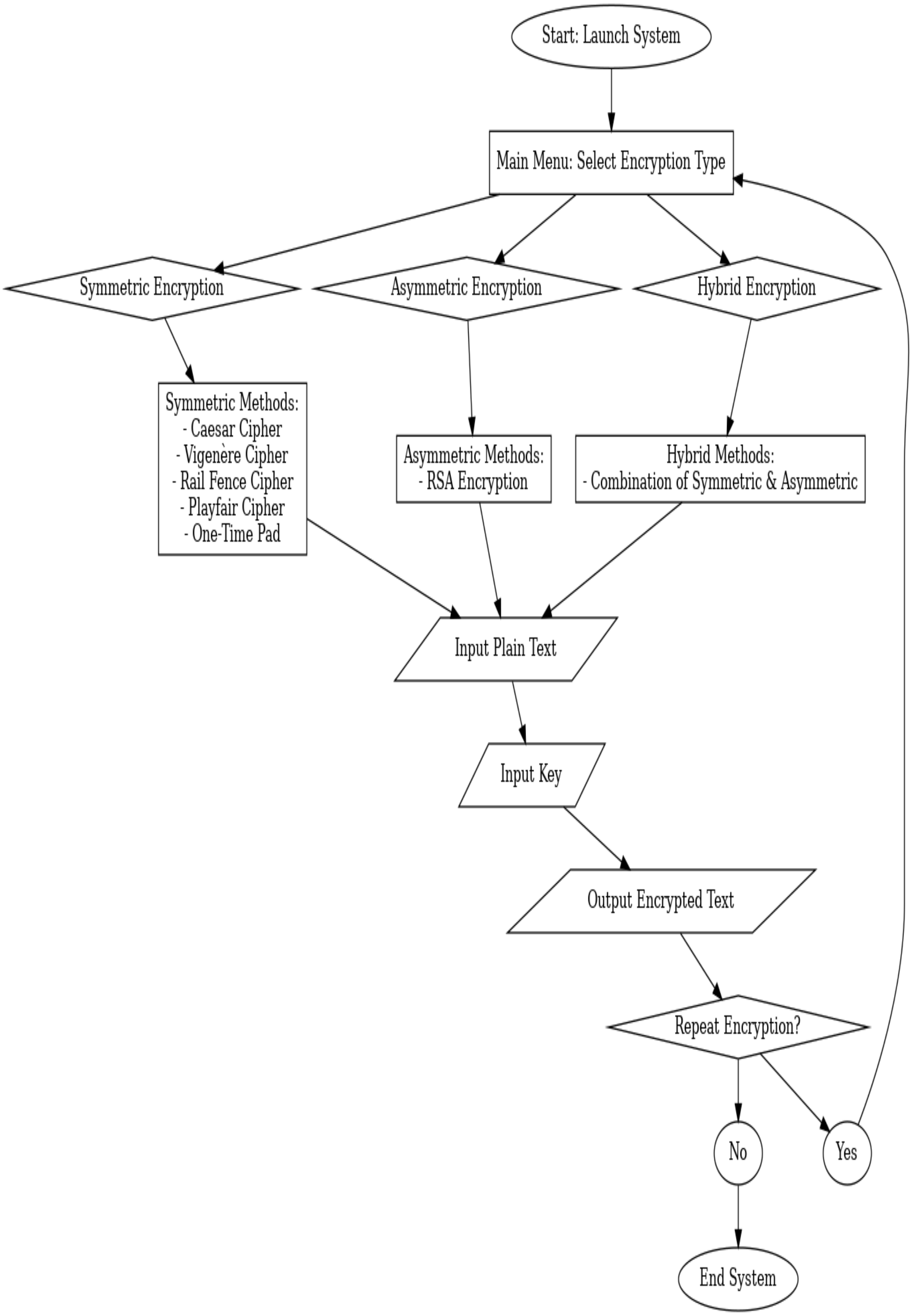
Advantages:

- Balances efficiency (symmetric) and security (asymmetric).
- Facilitates secure key exchange using asymmetric methods while encrypting data with faster symmetric methods.
- Suitable for modern communication protocols like HTTPS.

Disadvantages:

- Increases system complexity.
- Relies on proper implementation of both encryption types.
- Still vulnerable to attacks if one component is weak.

3. Flow Diagram



4. Code Snippets

```cpp
using namespace std;

// Simple symmetric encryption using XOR
string xorEncrypt(const string &data, char key)
{
    string encrypted = data;
    for (size_t i = 0; i < data.size(); i++)
    {
        // XOR with the key
        encrypted[i] ^= key;
    }
    return encrypted;
}

// Simple RSA-like key generation
pair<int, int> generateKeys()
{
    // Example prime number
    int p = 61;

    // Example prime number
    int q = 53;

    // Modulus for public and private keys
    int n = p * q;

    // Euler's totient function
    int phi = (p - 1) * (q - 1);

    // Public exponent (must be coprime with phi)
    int e = 17;

    // Private exponent (calculated using modular inverse of e mod phi)
    int d = 2753;

    // Return public key (e, n)
```

```cpp
pair<int, int> generateKeys()

    // Return public key (e, n)
    return {e, n};
}

// Simple RSA-like encryption (asymmetric encryption)
int rsaEncrypt(int message, int e, int n)
{
    int result = 1;
    for (int i = 0; i < e; i++)
    {
        // Modular exponentiation
        result = (result * message) % n;
    }
    return result;
}

// Simple RSA-like decryption
int rsaDecrypt(int cipher, int d, int n)
{
    int result = 1;
    for (int i = 0; i < d; i++)
    {
        // Modular exponentiation
        result = (result * cipher) % n;
    }
    return result;
}

// Convert string to integer
int stringToInt(const string &str)
{
    int result = 0;
    for (char c : str)
    {
        result = result * 256 + c;
```



```cpp
int stringToInt(const string &str)
        result = result * 256 + c;
    }
    return result;
}

// Convert integer back to string
string intToString(int num)
{
    string result;
    while (num > 0)
    {
        result = char(num % 256) + result;
        num /= 256;
    }
    return result;
}

// Function to encrypt the plaintext using Rail Fence Cipher
string railfenceencrypt(string plaintext, int key)
{
    if (key <= 1)
    {
        // No encryption for key <= 1
        return plaintext;

    // Create a vector of strings for each rail
    vector<string> rail(key);

    // 1 means moving down, -1 means moving up
    int direction = 1;
    int row = 0;

    // Fill the rails with characters in a zigzag pattern
    for (char ch : plaintext)
    {
```



```cpp
string railfenceencrypt(string plaintext, int key)

    // Fill the rails with characters in a zigzag pattern
    for (char ch : plaintext)
    {
        rail[row] += ch;
        row += direction;

        // Reverse direction when reaching the top or bottom rail
        if (row == 0 || row == key - 1)
        {
            direction *= -1;
        }
    }

    // Combine all rails into the ciphertext
    string ciphertext = "";
    for (const string &r : rail)
    {
        ciphertext += r;
    }

    return ciphertext;
}

// Function to decrypt the ciphertext using Rail Fence Cipher
string railfencedecrypt(string ciphertext, int key)
{
    if (key <= 1)
        // No decryption for key <= 1
        return ciphertext;

    // Track the number of characters in each rail
    vector<int> railLengths(key, 0);

    // 1 means moving down, -1 means moving up
    int direction = 1;
```

C: > Users > Admin > Desktop > ISPrjct.cpp > main()

```cpp
string railfencedecrypt(string ciphertext, int key)
    // 1 means moving down, -1 means moving up
    int direction = 1;
    int row = 0;

    // Calculate the length of each rail based on the zigzag pattern
    for (char ch : ciphertext)
    {
        railLengths[row]++;
        row += direction;

        if (row == 0 || row == key - 1)
        {
            direction *= -1;
        }
    }

    // Fill the rails with characters from the ciphertext
    vector<string> rail(key);
    int index = 0;
    for (int i = 0; i < key; i++)
    {
        rail[i] = ciphertext.substr(index, railLengths[i]);
        index += railLengths[i];
    }

    // Reconstruct the plaintext by following the zigzag pattern
    string plaintext = "";
    row = 0;
    direction = 1;

    for (size_t i = 0; i < ciphertext.length(); i++)
    {
        // Take the first character of the current rail
        plaintext += rail[row][0];

        // Remove the used character
```

C: > Users > Admin > Desktop > ISPrjct.cpp > main()

```cpp
string railfencedecrypt(string ciphertext, int key)
        // Take the first character of the current rail
        plaintext += rail[row][0];

        // Remove the used character
        rail[row] = rail[row].substr(1);
        row += direction;

        if (row == 0 || row == key - 1)
        {
            direction *= -1;
        }
    }

    return plaintext;
}

// Function to generate the full key based on the plaintext length
string generateKey(string plaintext, string key)
{
    int textLength = plaintext.length();
    int keyLength = key.length();
    string fullKey = key;

    // Repeat the key until it matches the length of the plaintext
    for (int i = 0; fullKey.length() < textLength; i++)
    {
        fullKey += key[i % keyLength];
    }

    return fullKey;
}

// Function to encrypt the plaintext using Vigenère Cipher
string vigencrypt(string plaintext, string key)
{
```



```cpp
string vigencrypt(string plaintext, string key)
{
    string ciphertext = "";
    key = generateKey(plaintext, key);

    for (size_t i = 0; i < plaintext.length(); i++)
    {
        if (isalpha(plaintext[i]))
        {
            char base = islower(plaintext[i]) ? 'a' : 'A';
            char shift = islower(key[i]) ? key[i] - 'a' : key[i] - 'A';
            ciphertext += (plaintext[i] - base + shift) % 26 + base;
        }
        else
        {
            ciphertext += plaintext[i];
        }
    }

    return ciphertext;
}

// Function to decrypt the ciphertext using Vigenère Cipher
string vigdecrypt(string ciphertext, string key)
{
    string plaintext = "";
    key = generateKey(ciphertext, key);

    for (size_t i = 0; i < ciphertext.length(); i++)
    {
        if (isalpha(ciphertext[i]))
        {
            char base = islower(ciphertext[i]) ? 'a' : 'A';
            char shift = islower(key[i]) ? key[i] - 'a' : key[i] - 'A';
            plaintext += (ciphertext[i] - base - shift + 26) % 26 + base;
        }
        else
```

```cpp
string vigdecrypt(string ciphertext, string key)
        }
        else
        {
            plaintext += ciphertext[i];
        }
    }

    return plaintext;
}

// Function to encrypt the plaintext using Caesar Cipher
string caesar_encrypt(string text, int shift)
{
    string result = "";
    for (char ch : text)
    {
        if (isalpha(ch))
        {
            char base = islower(ch) ? 'a' : 'A';
            result += (ch - base + shift) % 26 + base;
        }
        else
        {
            result += ch;
        }
    }
    return result;
}

// Function to decrypt the ciphertext using Caesar Cipher
string caesar_decrypt(string text, int shift)
{
    // Reverse the shift for decryption
    return caesar_encrypt(text, 26 - (shift % 26));
}
```


C: > Users > Admin > Desktop > ISPrjct.cpp > main()

```cpp
string caesar_decrypt(string text, int shift)
}

// Helper function to calculate determinant of a 2x2 matrix
int determinant(int matrix[2][2])
{
    return (matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]);
}

// Helper function to find the adjugate of a 2x2 matrix
void adjugate(int matrix[2][2], int adj[2][2])
{
    adj[0][0] = matrix[1][1];
    adj[0][1] = -matrix[0][1];
    adj[1][0] = -matrix[1][0];
    adj[1][1] = matrix[0][0];
}

// Function to compute modular inverse of a number under modulo m for rsa
int modInverse(int a, int m)
{
    a = a % m;
    for (int x = 1; x < m; x++)
        if ((a * x) % m == 1)
            return x;

    // Inverse does not exist
    return -1;
}

// Efficient modular exponentiation
int modularExponentiation(int base, int exp, int mod)
{
    int result = 1;

    // Reduce base if it's greater than mod
    base = base % mod;
```

C: > Users > Admin > Desktop > ISPrjct.cpp > main()

```cpp
int modularExponentiation(int base, int exp, int mod)
    int result = 1;

    // Reduce base if it's greater than mod
    base = base % mod;
    while (exp > 0)
    {
        // If exponent is odd, multiply the base with the result
        if (exp % 2 == 1)
        {
            result = (result * base) % mod;
        }
        // Square the base and halve the exponent
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

// Function to generate a random key of the same length as the plaintext
string generateRandomKey(const string &plaintext)
{
    string key = "";

    // Random number generator
    mt19937 gen(static_cast<unsigned long>(time(nullptr)));

    // Generate random values in the byte range (0-255)
    uniform_int_distribution<int> dist(0, 255);

    for (size_t i = 0; i < plaintext.length(); ++i)
    {
        // Random number Generator
        key += static_cast<char>(dist(gen));
    }

    return key;
```

C: > Users > Admin > Desktop > ISPrjct.cpp > main()

```cpp
}

// Function to encrypt the plaintext using the OTP
string encrypt(const string &plaintext, const string &key)
{
    string ciphertext = "";

    for (size_t i = 0; i < plaintext.length(); ++i)
    {
        // XOR each character of the plaintext with the key
        ciphertext += plaintext[i] ^ key[i];
    }
    return ciphertext;
}

// Function to decrypt the ciphertext using the OTP
string decrypt(const string &ciphertext, const string &key)
{
    string decryptedText = "";

    for (size_t i = 0; i < ciphertext.length(); ++i)
    {
        // XOR again to get the original plaintext
        decryptedText += ciphertext[i] ^ key[i];
    }

    return decryptedText;
}
// Function to compute the greatest common divisor (gcd)
int gcd(int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b, a % b);
}
```

C: > Users > Admin > Desktop > ISPrjct.cpp > main()

```cpp
int encrypt(int message, int e, int n)
{
    int result = 1;
    for (int i = 0; i < e; i++)
    {
        result = (result * message) % n;
    }
    return result;
}

// Function to decrypt a ciphertext (integer) using RSA
int decrypt(int ciphertext, int d, int n)
{
    int result = 1;
    for (int i = 0; i < d; i++)
    {
        result = (result * ciphertext) % n;
    }
    return result;
}

// Function to generate RSA keys
void generateRSAKeys(int &n, int &e, int &d, int p, int q)
{
    // Calculate n = p * q
    n = p * q;

    // Calculate Euler's totient function phi(n) = (p-1) * (q-1)
    int phi_n = (p - 1) * (q - 1);

    // Choose e such that 1 < e < phi(n) and gcd(e, phi(n)) = 1

    // Common choice for e is 17, it satisfies gcd(e, phi(n)) = 1
    e = 17;

    // Calculate d such that (d * e) % phi(n) = 1 (d is modular inverse of e
```


C: > Users > Admin > Desktop > ISPrjct.cpp > main()

```cpp
void generateRSAKeys(int &n, int &e, int &d, int p, int q)
    // Calculate d such that (d * e) % phi(n) = 1 (d is modular inverse of e
    d = modInverse(e, phi_n);
}

// Function to convert a string to a vector of integers (based on ASCII value
vector<int> stringToIntVector(const string &message)
{
    vector<int> result;
    for (char c : message)
    {
        result.push_back(static_cast<int>(c)); // Convert each character to i
    }
    return result;
}

// Function to convert a vector of integers back to a string
string intVectorToString(const vector<int> &intVector)
{
    string result;
    for (int i : intVector)
    {
        // Convert each integer back to a character
        result.push_back(static_cast<char>(i));
    }
    return result;
}
// Function to prepare the key matrix
void generateKeyMatrix(string key, char keyMatrix[5][5])
{
    vector<bool> used(26, false);

    // Treat 'J' and 'I' as the same
    used['J' - 'A'] = true;

    int row = 0, col = 0;
```

C: > Users > Admin > Desktop > ISPrjct.cpp > main()

```cpp
void generateKeyMatrix(string key, char keyMatrix[5][5])
    int row = 0, col = 0;

    for (char c : key)
    {
        c = toupper(c);
        if (!used[c - 'A'] && isalpha(c))
        {
            keyMatrix[row][col++] = c;
            used[c - 'A'] = true;
            if (col == 5)
            {
                col = 0;
                row++;
            }
        }
    }

    for (char c = 'A'; c <= 'Z'; ++c)
    {
        if (!used[c - 'A'])
        {
            keyMatrix[row][col++] = c;
            used[c - 'A'] = true;
            if (col == 5)
            {
                col = 0;
                row++;
            }
        }
    }
}

// Function to prepare the plaintext for Playfair cipher rules
string prepareText(string text)
{
    string prepared = "";
```


C: > Users > Admin > Desktop > ISPrjct.cpp > main()

```cpp
string prepareText(string text)
{
    string prepared = "";
    for (char c : text)
    {
        if (isalpha(c))
        {
            c = toupper(c);
            if (c == 'J')

                // Replace 'J' with 'I'
                c = 'I';
            prepared += c;
        }
    }

    for (size_t i = 0; i < prepared.length() - 1; i += 2)
    {
        if (prepared[i] == prepared[i + 1])
        {
            // Add filler 'X' for duplicate letters
            prepared.insert(i + 1, "X");
        }
    }
    if (prepared.length() % 2 != 0)
    {
        // Add filler 'X' if length is odd
        prepared += "X";
    }
    return prepared;
}

// Function to find position of a character in the key matrix
void findPosition(char keyMatrix[5][5], char c, int &row, int &col)
{
    for (int i = 0; i < 5; i++)
    {
```

C: > Users > Admin > Desktop > ISPrjct.cpp > main()

```cpp
void findPosition(char keyMatrix[5][5], char c, int &row, int &col)
    {
        for (int j = 0; j < 5; j++)
        {
            if (keyMatrix[i][j] == c)
            {
                row = i;
                col = j;
                return;
            }
        }
    }
}

// Function to encrypt plaintext using Playfair cipher
string encrypt(string plaintext, char keyMatrix[5][5])
{
    plaintext = prepareText(plaintext);
    string ciphertext = "";

    for (size_t i = 0; i < plaintext.length(); i += 2)
    {
        char first = plaintext[i];
        char second = plaintext[i + 1];
        int r1, c1, r2, c2;

        findPosition(keyMatrix, first, r1, c1);
        findPosition(keyMatrix, second, r2, c2);

        if (r1 == r2)
        {
            // Same row
            ciphertext += keyMatrix[r1][(c1 + 1) % 5];
            ciphertext += keyMatrix[r2][(c2 + 1) % 5];
        }
        else if (c1 == c2)
```

Picture 1:

xorEncrypt(const string &data, char key)

- Performs simple symmetric encryption using the XOR operation.
- Iterates through each character of the input string data and applies XOR with the given key.
- Returns the XOR-encrypted string.

generateKeys()

- Generates RSA-like public and private keys for encryption and decryption.
- Uses two example prime numbers p = 61 and q = 53.
- Calculation procedure:
    - Modulus n = p * q.
    - Euler's totient function phi = (p-1) * (q-1).
    - Public exponent e = 17 (usually used in RSA and must be coprime to phi).
    - Private exponent d = 2753 (modular inverse of e mod phi).
- Returns a pair of integers representing the public key (e, n).

rsaEncrypt(int message, int e, int n)

- Performs RSA-like encryption using modular exponentiation.
- Encrypts the message by raising it to the power of e and taking modulo n.
- Returns the encrypted result.

rsaDecrypt(int cipher, int d, int n)

- Performs RSA-like decryption using modular exponentiation.
- Decrypts the cipher by raising it to the power of d and taking modulo n.
- Returns the decrypted message.

stringToInt(const string &str)

- Converts a string to an integer.
- Iterates through each character in the string, updating the result as result = result * 256 + c (treating the string as a base-256 number).
- Returns the integer representation of the string.

Picture 2:

1. **intToString(int num)**
    - Converts an integer back into a string.
    - Iteratively extracts each byte (by using modulo 256) from the integer and prepends it to the resulting string.
    - Returns the string representation of the integer.
2. **railfenceencrypt(string plaintext, int key)**
    - Encrypts plaintext using the Rail Fence Cipher algorithm.
    - If the key <= 1, returns the plaintext without encryption.
    - Creates a vector of strings (rail) to represent the zigzag pattern for the cipher.
    - Traverses the plaintext:

        - Places each character into the appropriate "rail" based on the current row and direction (up or down).
        - Reverses the direction when the top or bottom rail is reached.
    - Combines all the rails into a single ciphertext string.
    - Returns the encrypted ciphertext.
3. **railfencedecrypt(string ciphertext, int key)**
    - Decrypts a ciphertext encrypted using the Rail Fence Cipher.
    - If the key <= 1, returns the ciphertext without decryption.
    - Sets up structures to track rail lengths and their positions to reconstruct the zigzag traversal.
    - Logic for decoding is likely implemented in the subsequent portion.

Picture 3:

railfencedecrypt(string ciphertext, int key)

- Decrypts ciphertext encrypted with the Rail Fence Cipher algorithm.
- Calculates the length of each rail based on the zigzag pattern.
- Fills the rails with characters from the ciphertext according to their lengths.
- Reconstructs the plaintext by following the zigzag traversal used during encryption.
- Returns the decrypted plaintext.

generateKey(string plaintext, string key)

- Generates a full key for use in the Vigenère Cipher by repeating the given key until its length matches the length of the plaintext.
- Takes the plaintext and key as input.
- Returns the full key as a string.

vigencrypt(string plaintext, string key)

- Encrypts plaintext using the Vigenère Cipher.
- Likely iterates through the plaintext, combining each character with the corresponding character in the key (based on the Vigenère Cipher logic).
- Generates key from plain text and string key
- Returns the encrypted plaintext.

Picture 4:

vigdecrypt(string plaintext, string key)

- Decrypts plain text with the Vigenère Cipher algorithm.
- Returns the decrypted text

caeser_encrypt(string plaintext, string key)

- Encrypts plaintext using the Caesar Cipher.
- Stores result in string
- Checks for capital letters in for loop and in IF condition
- Returns result which is encrypted text after iteration of for loop

caeser_decrypt(string plaintext, string key)

- Decrypts plaintext using the reverse of Caesar Cipher.

Picture 5:

determinant function:

- Computes the determinant of a 2x2 matrix.
- Formula used: det(A) = (a*d - b*c) for matrix [[a, b], [c, d]].

adjugate function:

- Calculates the adjugate (adjoint) of a 2x2 matrix.
- It swaps the diagonal elements and negates the off-diagonal elements of the input matrix.

modInverse function:

- Computes the modular multiplicative inverse of a modulo m.
- Uses a brute-force method to find an x such that (a * x) % m == 1.
- Returns -1 if the modular inverse does not exist.modular

Exponentiation function:

- Implements efficient modular exponentiation.
- Computes (base^exp) % mod using the square-and-multiply method to handle large exponents efficiently.

generateRandomKey function:

- Generates a random string (key) of the same length as the input plaintext.
- Uses a random number generator to create random characters from the byte range (0-255).
- Builds the key by appending random characters iteratively.

Picture 6:

encrypt function:

- Implements one-time pad (OTP) encryption.
- XORs each character in the plaintext with the corresponding character in the key.
- Returns the resulting ciphertext as a string.

decrypt function:

- Reverses OTP encryption by XORing each character of the ciphertext with the corresponding character in the key.
- Returns the original plain text.

gcd function:

- By usage of the Euclidean Algorithm this function finds the GCD of two integers.
- Uses recursion for simplicity.

encrypt function (RSA):

- Encrypts an integer message using RSA encryption.
- Performs modular exponentiation (message^e) % n.

decrypt function (RSA):

- Decrypts an integer ciphertext using RSA decryption.
- Computes (ciphertext^d) % n.

generateRSAKeys function:

- Generates RSA public and private keys.
- Calculates: n = p * q (product of two primes).
- Euler's totient function phi_n = (p-1) * (q-1).
- Choose e such that 1 < e < phi_n and gcd(e, phi_n) == 1.
- Computes the private key d, the modular inverse of e modulo phi_n.

Picture 7:

stringToIntVector Function:

- Converts a string into a vector of integers based on ASCII values.
- Iterates over each character in the string, casting it to its integer value.
- Returns a vector of integers.

intVectorToString Function:

- Converts a vector of integers back into a string.
- Iterates over each integer in the vector, casting it back to a character.
- Returns the resulting string.

generateKeyMatrix Function:

- Generates a 5x5 key matrix for the Playfair cipher based on a given key.
- Uses a used array to track which characters are included in the matrix.
- Treats 'J' and 'I' as the same character (a common Playfair cipher rule).
- Fills the matrix row by row, first with characters from the key, then with the remaining letters of the alphabet (excluding duplicates).

Picture 8:

prepareText function

- Converts all alphabetic characters in the input to uppercase.
- Replaces the letter 'J' with 'I' as per Playfair cipher convention.
- Detects consecutive duplicate letters in the input and inserts an 'X' between them.

- If the length of the prepared text is odd, adds an 'X' at the end to make the length even.
- Output: Returns the processed text ready for encryption.

findPosition function

- Finds the row and column indices of a given character in the Playfair cipher's key matrix.
- Iterates through the 5x5 key matrix to locate the character.
- Updates the references for row (row) and column (col) with the found indices.

encrypt function

- Prepares the plaintext using the prepareText function.
- Processes the prepared text two characters at a time:
- Uses the findPosition function in order to obtain the positions of the two characters in the key matrix.
- Applies Playfair cipher rules:If both characters are in the same row then the cipher replaces each with the character to its right in the same row if it is not full it will circle through to the first element.
- If both characters are in the same column, it substitutes each with the character below it (wrapping to the top if needed).
- If neither of the above, substitutes the characters with the ones at the opposite corners of the rectangle they form in the matrix.

5. Time Complexities

1. Symmetric Encryption

- Caesar Cipher:
    - *Time Complexity:* O(n), where n is the length of the plaintext.
    - *Explanation:* Each character is shifted a fixed number of positions, which is a constant-time operation for each character.
- Vigenère Cipher:
    - *Time Complexity:* O(n), where n is the length of the plaintext.
    - Explanation: Each character is shifted based on a key, but the shift value changes periodically. This still results in a linear time complexity.
- Rail Fence Cipher:
    - Time Complexity: O(n), where n is the length of the plaintext.

    - Explanation: The plaintext is written diagonally across a grid, and then read off row by row. This process involves iterating through the plaintext once and performing simple operations, leading to linear time complexity.
- Playfair Cipher:
    - Time Complexity: O(n), where n is the length of the plaintext.
    - Explanation: Each pair of plaintext letters is encrypted using a 5x5 matrix. While the lookup and substitution operations might seem complex, they are performed a constant number of times for each plaintext pair, resulting in linear time complexity.
- One-Time Pad (OTP):
    - Time Complexity: O(n), where n is the length of the plaintext.
    - Explanation: Each plaintext bit is XORed with a corresponding random key bit. This is a simple bitwise operation that takes constant time per bit, leading to linear time complexity.

2. Asymmetric Encryption (RSA-like)

- Key Generation:
    - Time Complexity: O(k^3), where k is the key size in bits.
    - Explanation: Key generation involves finding large prime numbers and performing modular exponentiation, which are computationally intensive operations.
- Encryption:
    - Time Complexity: O(k^3), where k is the key size in bits.
    - Explanation: Encryption involves modular exponentiation, which is the dominant operation.
- Decryption:
    - Time Complexity: O(k^3), where k is the key size in bits.
    - Explanation: Similar to encryption, decryption involves modular exponentiation.

3. Hybrid Encryption

- Time Complexity: It depends on the specific symmetric and asymmetric algorithms used.
    - Explanation: Hybrid encryption involves both symmetric and asymmetric operations. The overall time complexity will depend on the efficiency of the chosen algorithms.

6. Best Performing Encryption Technique

Out of all the ciphering techniques used in our code, the most efficient out of all of them is perhaps the Caesar Cipher.

The following is the justification for our chosen technique:

Why Caesar Cipher is Most Efficient:

Time Complexity:

Caesar Cipher operates in O(n). It performs a simple character shift operation, requiring minimal operations of computation.

Performance:

It involves straightforward arithmetic operations on characters, making it extremely fast, especially for small to medium-sized text.

Coding Mechanisms:

The implementation is simple, as seen in functions like caesar_encrypt. It uses a loop to iterate through the plaintext, applying the shift, with no need for additional data structures or computational overhead.

Comparison with Other Ciphers:

Vigenère Cipher:

It has the same time complexity, but slightly more complex due to the use of a keyword and handling multiple character shifts.

Rail Fence Cipher:

It has the same time complexity, but the zigzag pattern introduces additional overhead in managing positions.

Playfair Cipher:

This requires preprocessing to generate the key matrix and handle other functions, adding to complexity compared to Caesar.

RSA Encryption:

It has the highest time complexity due to modular arithmetic and is slower than all classical ciphers.

Final Verdict

The Caesar Cipher is the most efficient in terms of simplicity, speed, and coding mechanisms but, its efficiency comes at the cost of security, as it is highly vulnerable to attacks by third parties.

If efficiency is the primary criterion, Caesar Cipher stands out; for practical applications, more secure ciphers like Vigenère or hybrid systems should be considered.

7. Key Achievements of Project

1. **Algorithmic Diversity**
    - Related operations have been successfully performed with many different encryption algorithms.
    - Showed similar and distinguishable methods of encryption
    - Demonstration on the adaptability of crypto techniques
2. **Educational Significance**
    - Offered completely operational, first-hand training opportunity
    - Connected principles of the existence of cryptography at the theoretical level with examples of its use
    - Provided understandings into mathematical background of encryption
3. **Technical Complexity**
    - Created the open, parameterizable and scalable cryptographic architecture
    - Put in place high level of encryption and decryption functions
    - Showed good programming practices in C++

8. Future Improvements and Recommendations

While the current implementation serves as an excellent educational tool, potential enhancements could include:

- Improving on the user interface and improved error checking
- Enabling of large messages since certain clients cannot handle large blocks of data, support for this kind of sizes is needed.
- Setting up a GUI based program which could be intricately integrated with softwares and used on an as needed basis.

For future developers and researchers:

- Keep abreast with the market trend with regards to encryption technologies.
- Consequently, there is a need to extend increased attention towards developing better and, conversely, safer use of communication.
- Cryptographic studies remain a very dynamic field, and such endeavours go along ways towards enhancing an understanding and usage of improved forms of secure communication systems.

9. Final Thoughts

Cryptography remains a critical field in our increasingly digital world. This project underscores the importance of understanding and implementing secure communication methods. By providing a hands-on exploration of encryption techniques, we contribute to the broader understanding of digital security and data protection.

The journey of cryptography is ongoing, with new challenges and innovations emerging constantly. This project serves as a stepping stone towards more advanced and secure communication systems.