**PANDIT DEENDAYAL ENERGY UNIVERSITY**

**SCHOOL OF TECHNOLOGY**



**Course: Information Security**

**Course Code: 20CP304P**

**LAB MANUAL**

**B.Tech. (Computer Science and Engineering)**

**Semester 5**

|  |  |
| --- | --- |
| **Submitted To:**  Dr. Hargeet Kaur | **Submitted By:**  Aditya Jethani  21BCP166  G-5 batch |

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **List of experiments** | **Date** | **Sign** |
| **1** | Download and Practise Cryptool. | 5/8/23 |  |
| **2** | Study and implement Caesar cipher with encryption, decryption. | 5/8/23 |  |
| **3** | Study and implement a program for Transposition(Columnar) cipher to encrypt and decrypt the message . | 12/8/23 |  |
| **4** | Study and implement a program for Rail Fence Transposition Cipher to encrypt and decrypt the message . | 19/8/23 |  |
| **5** | Study and implement a program for Vigenere cipher to encrypt and decrypt the message . | 26/8/23 |  |
| **6** | Study and implement a program for 5\*5 Playfair cipher. | 2/9/23 |  |
| **7** | Implement N-gram Hill cipher. | 23/9/23 |  |
| **8** | Use Crypto++ library to implement encryption and decryption of different block ciphers. | 14/10/23 |  |
| **9** | To Study and Implement a program for RSA Algorithm to encrypt and decrypt the message. | 21/10/23 |  |
| **10** | To Study and Implement a program for Python code to use RSA for generation and verification of digital signature on file. | 21/10/23 |  |

**Experiment No : 1**

**Aim** : Download and Practise Cryptool.

**Introduction** : CrypTool is an [open-source](https://en.wikipedia.org/wiki/Open-source_software) project that is a free [e-learning](https://en.wikipedia.org/wiki/E-learning) software for illustrating [cryptographic and cryptanalytic concepts](https://en.wikipedia.org/wiki/Cryptography). According to "Hakin9",CrypTool is worldwide the most widespread e-learning software in the field of [cryptology](https://en.wikipedia.org/wiki/Cryptology).

They contain most [classical ciphers](https://en.wikipedia.org/wiki/Classical_cipher), as well as modern symmetric and [asymmetric cryptography](https://en.wikipedia.org/wiki/Public-key_cryptography) including [RSA](https://en.wikipedia.org/wiki/RSA_(algorithm)), [ECC](https://en.wikipedia.org/wiki/Elliptic_curve_cryptography), [digital signatures](https://en.wikipedia.org/wiki/Digital_signature), hybrid encryption, [homomorphic encryption](https://en.wikipedia.org/wiki/Homomorphic_encryption), and [Diffie–Hellman key exchange](https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange). Methods from the area of [quantum cryptography](https://en.wikipedia.org/wiki/Quantum_cryptography) (like [BB84 key exchange protocol](https://en.wikipedia.org/wiki/BB84)) and the area of [post-quantum cryptography](https://en.wikipedia.org/wiki/Post-quantum_cryptography) (like [McEliece](https://en.wikipedia.org/wiki/McEliece_cryptosystem), WOTS, [Merkle-Signature-Scheme](https://en.wikipedia.org/wiki/Merkle_signature_scheme), [XMSS, XMSS\_MT, and SPHINCS](https://en.wikipedia.org/wiki/Hash-based_cryptography)) are implemented. In addition to the algorithms, solvers (analyzers) are included, especially for classical ciphers. Other methods (for instance [Huffman code](https://en.wikipedia.org/wiki/Huffman_coding), [AES](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard), [Keccak](https://en.wikipedia.org/wiki/SHA-3), [MSS](https://en.wikipedia.org/wiki/Merkle_signature_scheme)) are visualized.

**1.Caeser Cipher**

The Caesar Cipher technique is one of the earliest and simplest methods of encryption technique. It’s simply a type of substitution cipher, i.e., each letter of a given text is replaced by a letter with a fixed number of positions down the alphabet. For example with a shift of 1, A would be replaced by B, B would become C, and so on. The method is apparently named after Julius Caesar, who apparently used it to communicate with his officials.

**2.Vigenere cipher**

Vigenere Cipher is a method of encrypting alphabetic text. It uses a simple form of polyalphabetic substitution. A polyalphabetic cipher is any cipher based on substitution, using multiple substitution alphabets. The table consists of the alphabets written out 26 times in different rows, each alphabet shifted cyclically to the left compared to the previous alphabet, corresponding to the 26 possible Cea sar Ciphers.At different points in the encryption process, the cipher uses a different alphabet from one of the rows.The alphabet used at each point depends on a repeating keyword.

**3.Vernam Cipher**

Vernam Cipher is a method of encrypting alphabetic text. It is one of the Substitution techniques for converting plain text into cipher text. We take a key to encrypt the plain text whose length should be equal to the length of the plain text.Assign a number to each character of the plain-text and the key according to alphabetical order. Bitwise XOR both the number (Corresponding plain-text character number and Key character number). Subtract the number from 26 if the resulting number is greater than or equal to 26, if it isn’t then leave it.

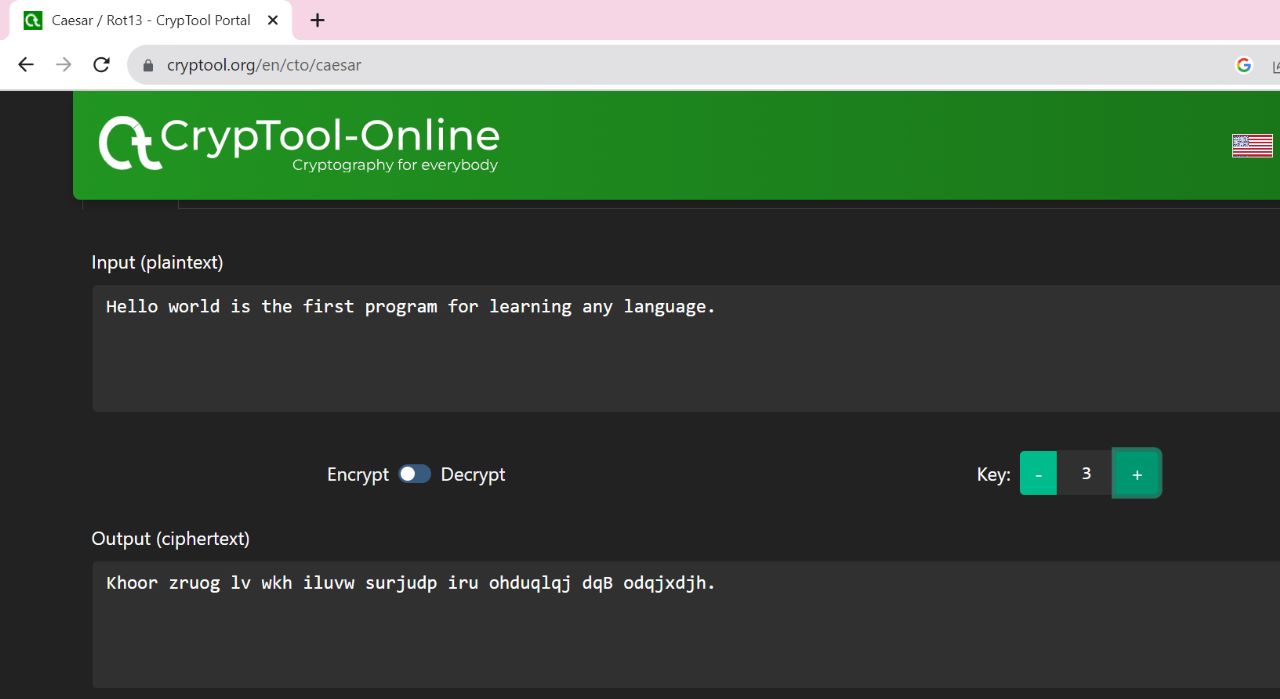
**4.Simple Columnar Transposition**

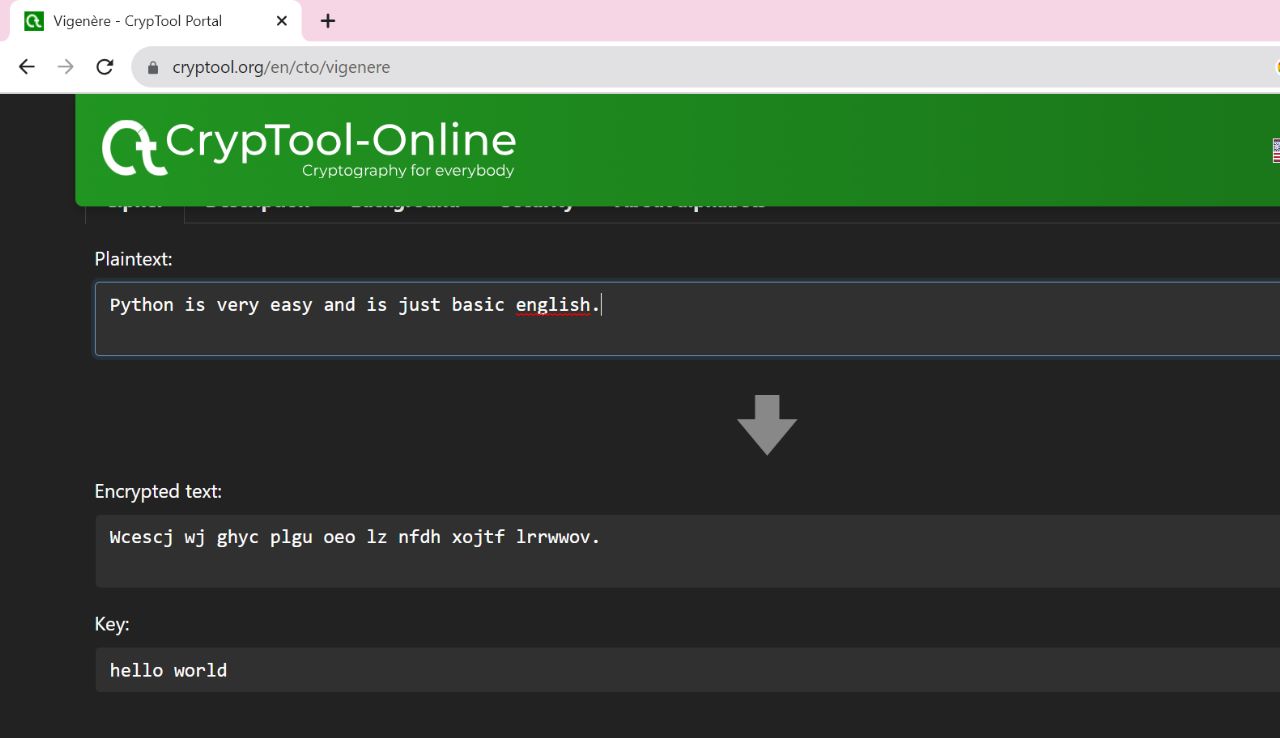
In a transposition cipher, the order of the alphabets is re-arranged to obtain the cipher-text.The message is written out in rows of a fixed length, and then read out again column by column, and the columns are chosen in some scrambled order.Width of the rows and the permutation of the columns are usually defined by a keyword.The permutation is defined by the alphabetical order of the letters in the keyword. Any spare spaces are filled with nulls or left blank or placed by a character .Finally, the message is read off in columns, in the order specified by the keyword.

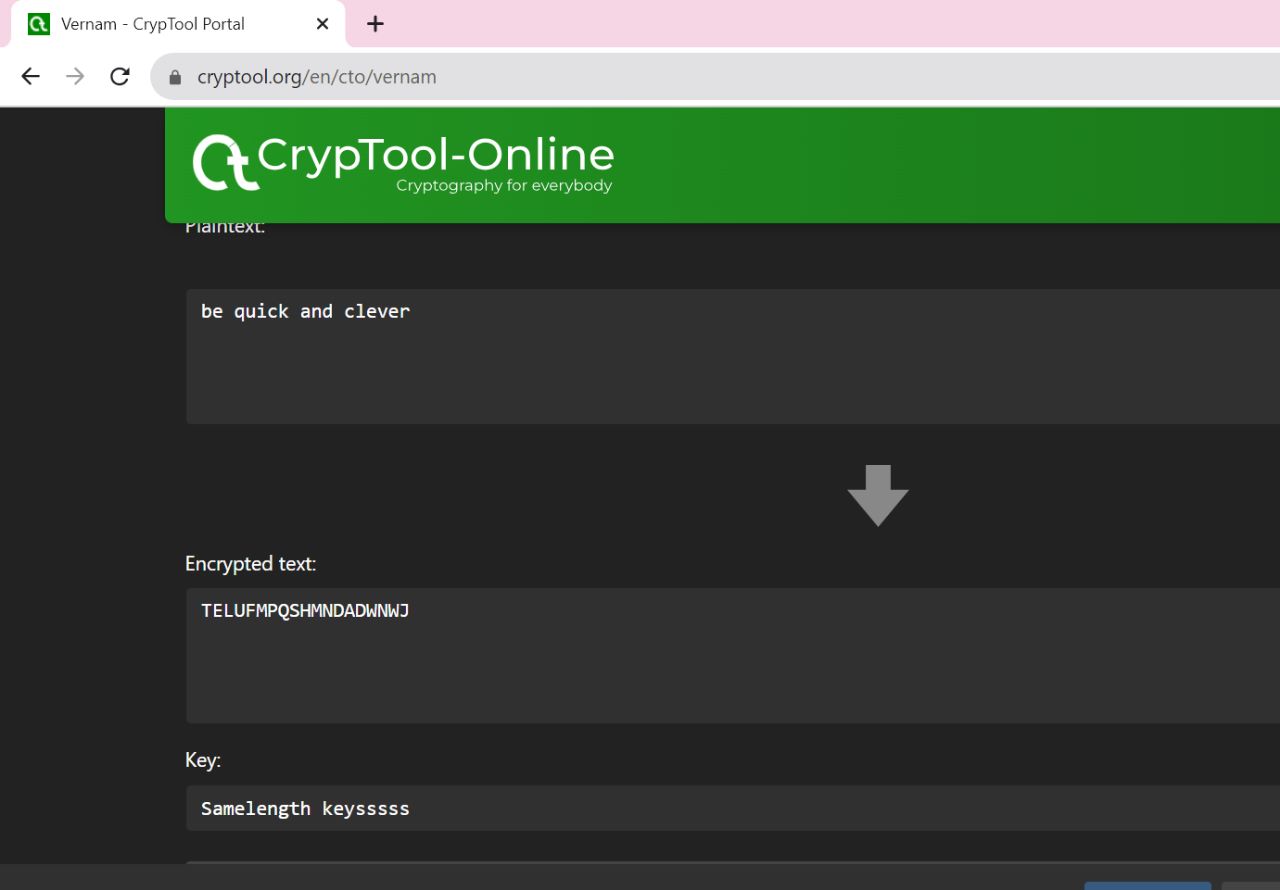
**5.Mono Alphabetic Substition**

A monoalphabetic cipher is any cipher in which the letters of the plain text are mapped to cipher text letters based on a single alphabetic key. Examples of monoalphabetic ciphers would include the Caesar-shift cipher, where each letter is shifted based on a numeric key, and the atbash cipher, where each letter is mapped to the letter symmetric to it about the center of the alphabet.

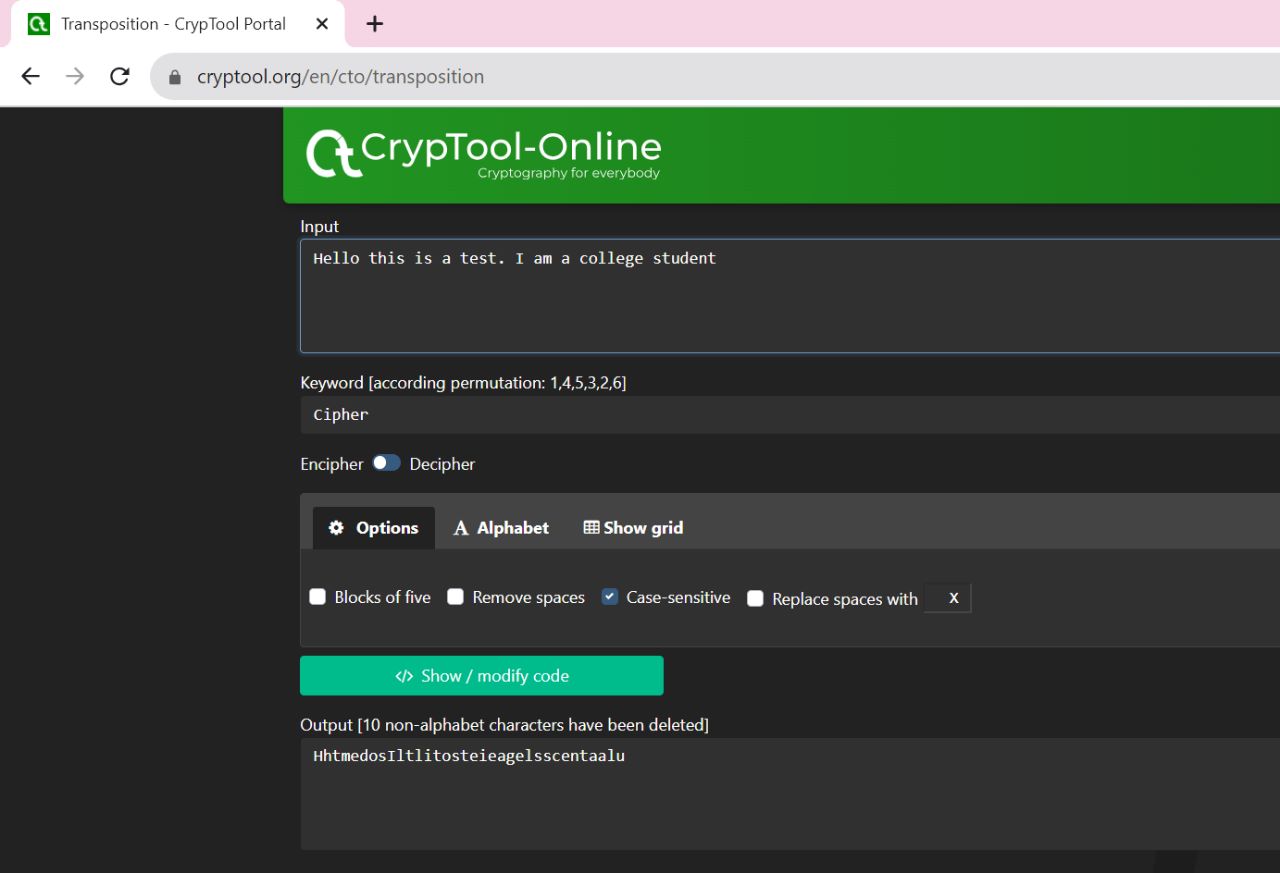
**Output:**

**Caeser Cipher** 

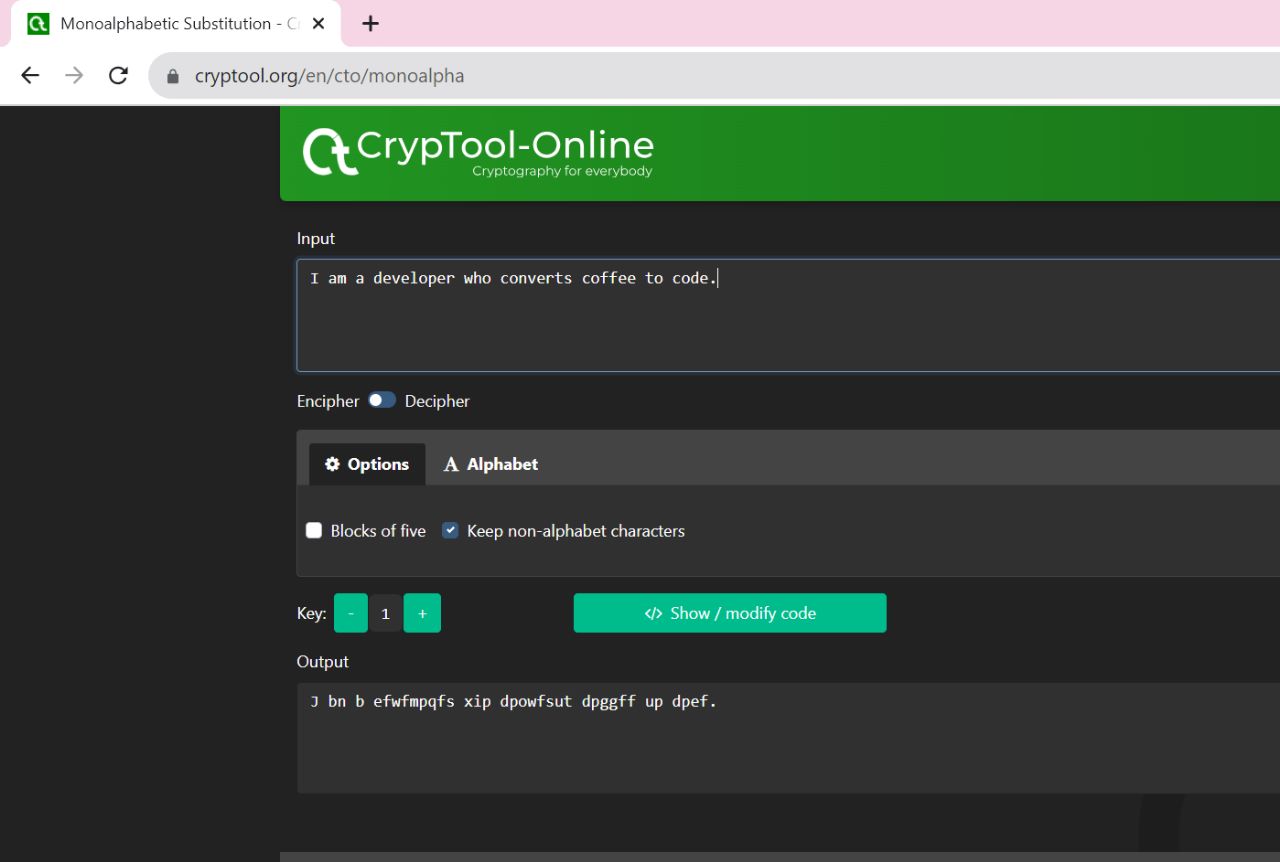
**Vigenere cipher**



**Vernam Cipher**



**Simple Columnar Transposition**



**Mono Alphabetic Substition**

**References :**

1.Wikipedia(https://en.wikipedia.org/wiki/CrypTool#:~:text=CrypTool%20is%20an%20open%2Dsource,in%20the%20field%20of%20cryptology.&text=CrypTool%20implements%20more%20than%20400%20algorithms.)

2. GeeksforGeeks(https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/)

**EXPERIMENT NO: 2**

**Aim:** Study and implement Caesar cipher with encryption, decryption.

**Introduction:** The Caesar cipher is one of the earliest and simplest forms of encryption. This is a type of substitution cipher, where each character in the original message is moved to a number of places below the line.

For example, if there were 3 changes, A would be replaced by D, B by E, C by F, and so on. The first messages will be saved as "HELLO" and "KHOOR".

To encrypt a message using a Caesar cipher:

* Select shift number - This is the number of positions you shift each character in the list. They are typically 3 or 4 inches.
* Write a simple text message.
* Replace each letter in the message with the number you selected. So if the shift is 3, then A becomes D, B E and so on.
* The resulting ciphertext is an encrypted message.

To dig into the Caesar Cipher:

* Displays the shift number originally used. Maybe they know this or maybe you should wonder. Take the cipher text and shift each character back to A by shift number. So if the variable was 3, then D would be A, E would be B, and so on.
* The resulting plaintext is the extracted original message.
* The Caesar cipher is easy to crack because there are only 25 possible changes. It does not offer the worst protection and is of particular historical interest today. However, it represents an early attempt to use alternatives to protect the message.

**Program Code:**

#include <iostream>

using namespace std;

string encryptMessage(const string message, int key) {

    string encryptedMessage = "";

    for (char ch : message) {

        if (islower(ch)) {

            ch = (ch - 'a' + key) % 26 + 'a';

        } else if (ch == ' ') {

            encryptedMessage += ch;

            continue;

        } else {

            cout << "Invalid Message\n";

            return "";

        }

        encryptedMessage += ch;

    }

    return encryptedMessage;

}

string decryptMessage(const string message, int key) {

    return encryptMessage(message, 26 - key);

}

int main() {

    string text;

    int key;

    cout << "Enter a message to encrypt: ";

    getline(cin, text);

    cout << "Enter the key: ";

    cin >> key;

    string encryptedText = encryptMessage(text, key);

    if (!encryptedText.empty()) {

        cout << "Encrypted message: " << encryptedText << endl;

        string decryptedText = decryptMessage(encryptedText, key);

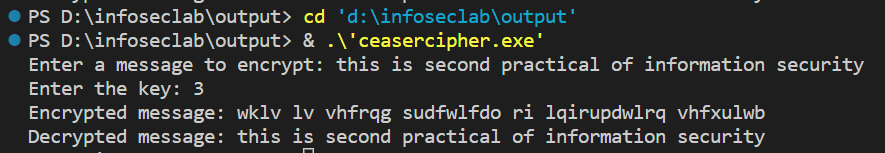
        cout << "Decrypted message: " << decryptedText << endl;

    }

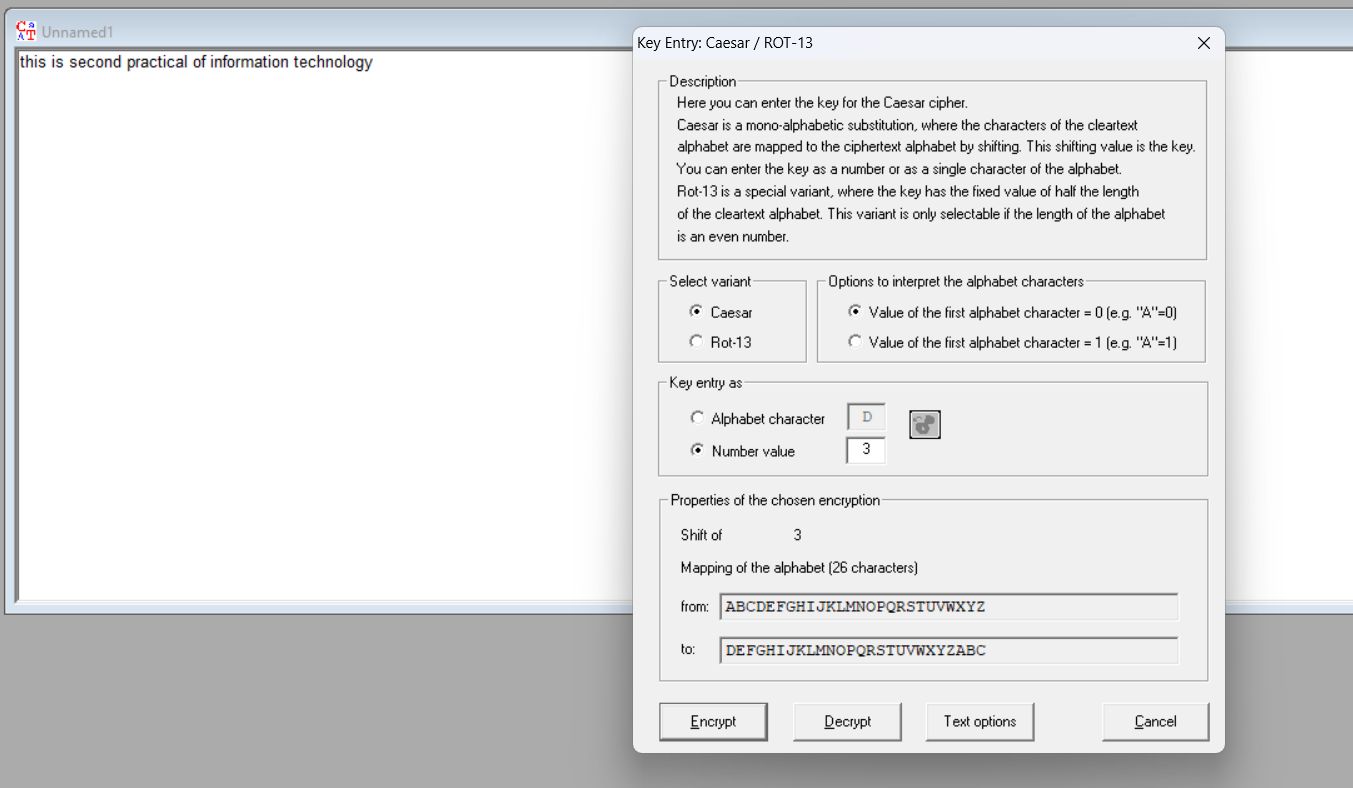
    return 0;

}

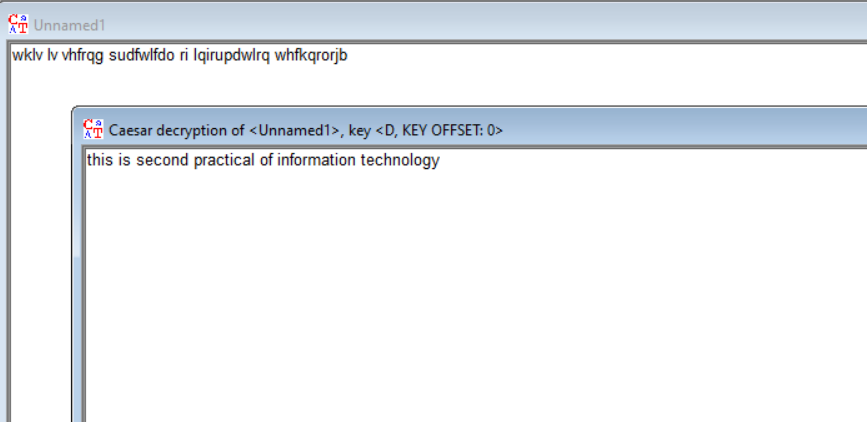
**Program Output(IDE):**

****

**Program Output (Cryptool):**





****

**Cryptanalysis:**

Time complexity:

The time complexity of the Caesar cipher is O(n), where 'n' is the length of the input message (the number of characters in the plaintext or ciphertext). The number of characters processed is directly proportional to the length of the input message, the time complexity is linear, or O(n). Regardless of the shift value, the time complexity remains the same because the same amount of work is done for each character.

Advantages:

* Simple to implement - it just requires substituting each letter with another shifted letter. Easy to do by hand.
* Fast to encrypt and decrypt a message.
* Provides a slight amount of obfuscation of the original message.

Disadvantages:

* Very weak security - there are only 25 possible shifts so it is easy to brute force and crack the cipher.
* The frequency analysis of the cipher text allows the cipher to be cracked easily. More common letters like 'E' and 'T' will still appear frequently.
* The shift is reusable and needs to be protected. If the shift is known, the cipher can be decrypted trivially.
* It only works well for English text, not for other languages or symbols.
* Does not provide confidentiality, integrity, and authenticity in a message.

**Applications:**

* As a simple method to add basic encryption to messages. For example, Julius Caesar is said to have used it to communicate with his generals.
* As a teaching tool to illustrate encryption concepts. It provides a simple substitution cipher to demonstrate core encryption principles.
* To obscure words or passages in media to avoid filters or spoiling plot points.
* To provide a layer of simple obscurity for things like website or product codes.

**References:**

* Wikipedia(https://en.wikipedia.org/wiki/Caesar\_cipher)
* GeeksforGeeks(https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/)

**EXPERIMENT NO: 3**

**Aim:** Study and implement a program for Transposition(Columnar) cipher to encrypt and decrypt the message .

**Introduction:** In a columnar transposition, the message is written out in rows of a fixed length, and then read out again column by column, and the columns are chosen in some scrambled order. Both the width of the rows and the permutation of the columns are usually defined by a keyword.

In a transposition cipher, the order of the alphabets is re-arranged to obtain the cipher-text.

* The message is written out in rows of a fixed length, and then read out again column by column, and the columns are chosen in some scrambled order.
* Width of the rows and the permutation of the columns are usually defined by a keyword.
* Any spare spaces are filled with nulls or left blank or placed by a character (Example: \_).
* Finally, the message is read off in columns, in the order specified by the keyword.

To decipher it, the recipient has to work out the column lengths by dividing the message length by the key length. Then they can write the message out in columns again, then re-order the columns by reforming the key word.

**Program Code:**

#include <bits/stdc++.h>

using namespace std;

string encryptSimpleColumnTransposition(const string& plainText, const string& key) {

    string encryptedText;

    int keyLength = key.length();

    int textLength = plainText.length();

    int numRows = (textLength + keyLength - 1) / keyLength;

    char matrix[numRows][keyLength];

    int index = 0;

    for (int i = 0; i < numRows; i++) {

        for (int j = 0; j < keyLength; j++) {

            if (index < textLength) {

                matrix[i][j] = plainText[index++];

            } else {

                matrix[i][j] = ' ';

            }

        }

    }

    string sortedKey = key;

    sort(sortedKey.begin(), sortedKey.end());

    int permutationIndex[keyLength];

    for (int i = 0; i < keyLength; i++) {

        permutationIndex[i] = key.find(sortedKey[i]);

    }

    for (int i = 0; i < keyLength; i++) {

        for (int j = 0; j < numRows; j++) {

            encryptedText += matrix[j][permutationIndex[i]];

        }

    }

    return encryptedText;

}

string decryptSimpleColumnTransposition(const string& encryptedText, const string& key) {

    string decryptedText;

    int keyLength = key.length();

    int textLength = encryptedText.length();

    int numRows = (textLength + keyLength - 1) / keyLength;

    char matrix[numRows][keyLength];

    string sortedKey = key;

    sort(sortedKey.begin(), sortedKey.end());

    int permutationIndex[keyLength];

    for (int i = 0; i < keyLength; i++) {

        permutationIndex[i] = key.find(sortedKey[i]);

    }

    int index = 0;

    for (int i = 0; i < keyLength; i++) {

        for (int j = 0; j < numRows; j++) {

            matrix[j][permutationIndex[i]] = encryptedText[index++];

        }

    }

    for (int i = 0; i < numRows; i++) {

        for (int j = 0; j < keyLength; j++) {

            decryptedText += matrix[i][j];

        }

    }

    return decryptedText;

}

int main() {

    string plainText;

    string key;

    cout << "Enter the plain text: ";

    getline(cin, plainText);

    cout << "Enter the key: ";

    getline(cin, key);

    string encryptedText = encryptSimpleColumnTransposition(plainText, key);

    cout << "Encrypted Text: " << encryptedText << endl;

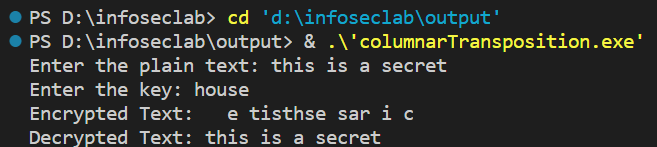
    string decryptedText = decryptSimpleColumnTransposition(encryptedText, key);

    cout << "Decrypted Text: " << decryptedText << endl;

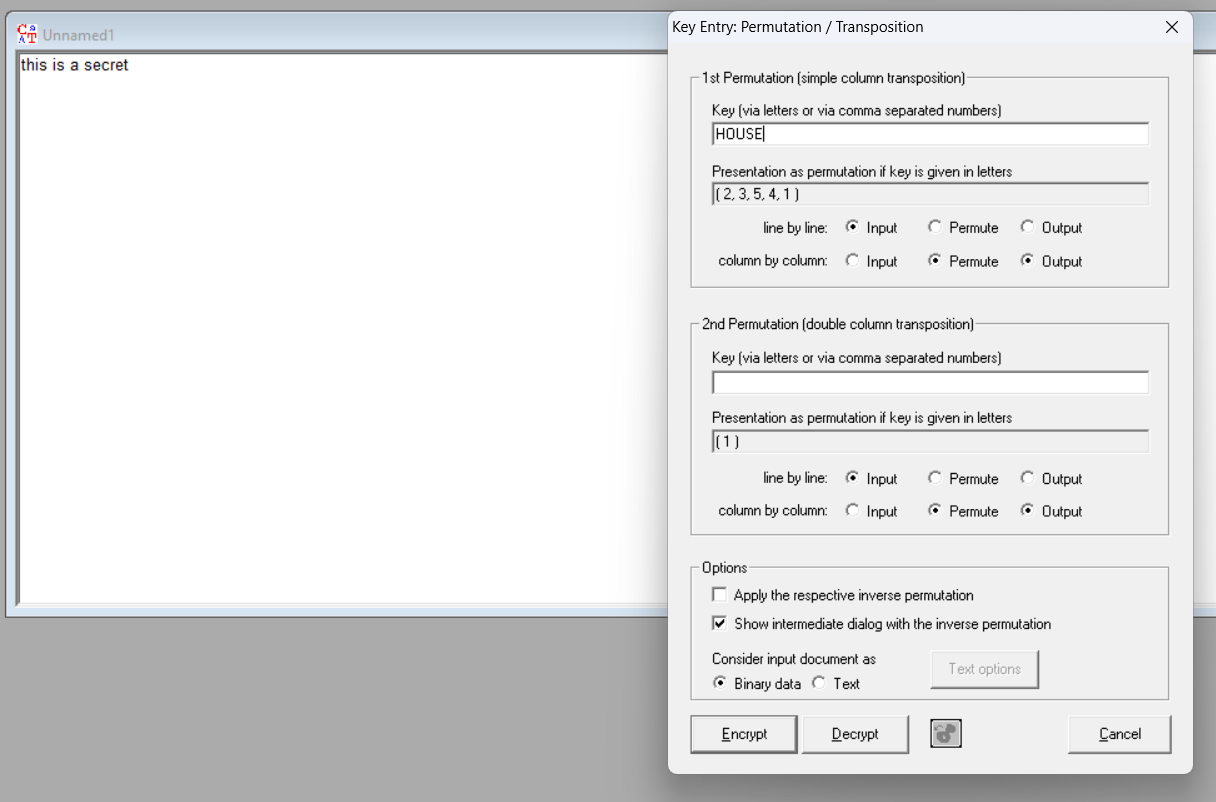
    return 0;

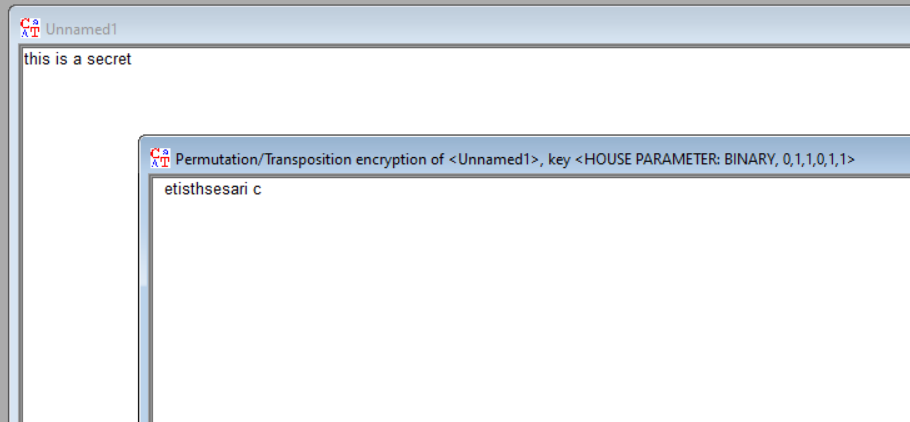
}

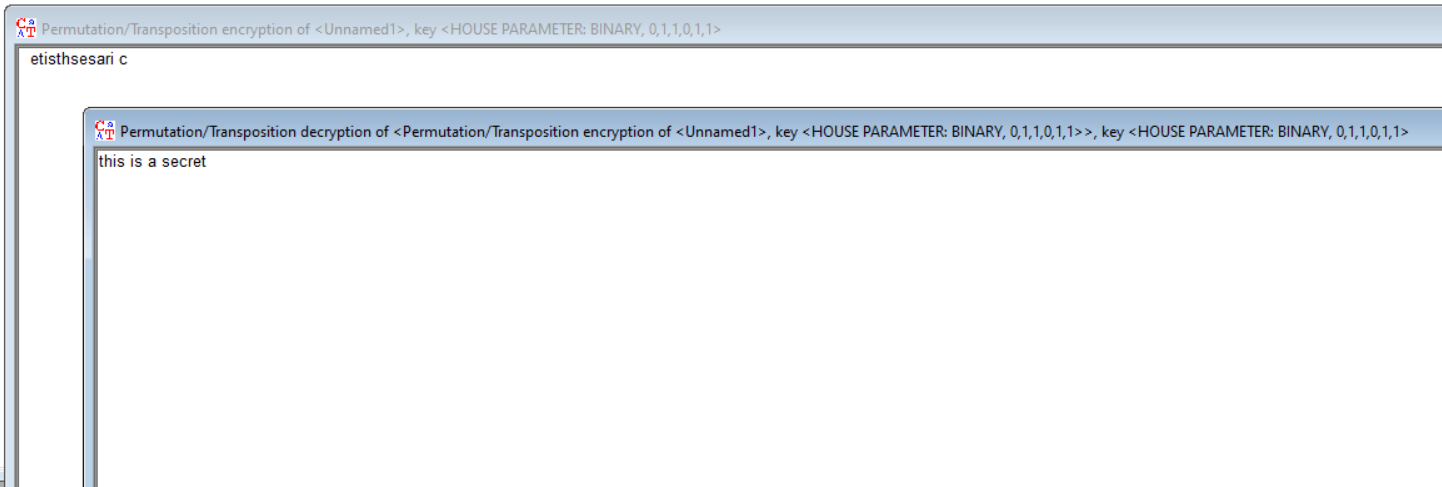
**Program Output(IDE):**

****

**Program Output (Cryptool):**

****

****

****

**Cryptanalysis:**

**Advantages of Columnar Transposition Cipher:**

* **Key Length:** Increasing the key length (number of columns) enhances the security of the cipher. Longer keys make brute force attacks more challenging.
* **Confusion :** The cipher exhibits confusion (hiding the relationship between plaintext and ciphertext) making it resistant to simple attacks.
* **Message Length Hiding:** The use of null characters or spaces to fill in the matrix hides the true length of the message, making it more challenging for attackers to determine the length of the original message.

**Disadvantages of Columnar Transposition Cipher:**

* **Vulnerable to Frequency Analysis:** While the cipher disrupts the frequency analysis of individual letters, longer repeated sequences or known plaintext can still provide enough information for frequency analysis to be effective.
* **Short Key Lengths:** If the key length is short, the number of possible key permutations is limited, making brute force attacks feasible.
* **Patterns in Ciphertext:** Depending on the key arrangement and original plaintext, there might be patterns in the ciphertext that attackers can exploit.

**Time Complexity:**

**Encrypt(encryptSimpleColumnTransposition):**

**Creating a matrix** requires iterating through each cell of the matrix, whose dimensions are **numRows** (indicated by message and key length) and **keyLength** (indicated by key length) This takes **O(numRows \* keyLength)** ) time.

**Formatting a key:** The key formation step takes **O(keyLength \* log(keyLength)**) time, which is the complex time for formatting keyLength objects.

**Creating an encrypted text:** Creating an encrypted text requires iterating through each column of the matrix (**keyLength** column), and for each column, to pass through **numRows** rows this takes **O(keyLength \* numRows)** ) time.

Overall, the main thing is that the matrix is ​​being built, which has a time complexity of **O(numRows \* keyLength).**

**Decrypt ( decryptSimpleColumnChange ):** . Building a matrix: Like encryption, it requires reconstruction of each matrix cell with **numRows** and **keyLength** dimensions, resulting in a time complexity of **O(numRows \* keyLength).**

**Key formation:** Key formation takes **O(keyLength \* log(keyLength)** time.

**Creating decrypted text:** Like encryption, creating decrypted text requires iterating through each row of the matrix (**numRows** rows), and in each row, passing through the **keyLength** columns takes **O(keyLength \* numRows)** time.

Overall, again, the key is building the matrix, which has a time complexity of **O(numRows \* keyLength).**

The time constraint for both encryption and decryption is the creation of the matrix, which takes **O(numRows \* keyLength)** time. Configuring the key and creating encryption or decryption also helps save time, but often.

**Applications:**

* **Steganography:** Steganography is the practice of concealing information within other, seemingly innocuous data. The columnar transposition cipher can be used to create steganographic messages by embedding a secret message within a larger text, making it less conspicuous.
* **Cryptography Education:** While not suitable for secure encryption in modern times, the columnar transposition cipher can be used as a stepping stone to teach basic cryptography concepts before moving on to more complex ciphers and encryption techniques.
* **Basic Message Privacy:** While not suitable for high-security scenarios, the columnar transposition cipher can provide a basic level of privacy for casual communication among individuals who are not well-versed in cryptography.
* **Puzzle and Games:** The cipher can be used in puzzles, escape room challenges, or treasure hunts. Players need to decrypt messages to advance in the game or solve puzzles.

**References:**

* Wikipedia(https://en.wikipedia.org/wiki/Transposition\_cipher)
* GeeksforGeeks(<https://www.geeksforgeeks.org/columnar-transposition-cipher/>

**EXPERIMENT NO: 4**

**Aim:** Study and implement a program for Rail Fence Transposition Cipher to encrypt and decrypt the message .

**Introduction:** The rail fence cipher (also called a zigzag cipher) is a form of transposition cipher. It derives its name from the way in which it is encoded. In a transposition cipher, the order of the alphabets is re-arranged to obtain the cipher-text.

Encryption

In the rail fence cipher, the plain-text is written downwards and diagonally on successive rails of an imaginary fence.

When we reach the bottom rail, we traverse upwards moving diagonally, after reaching the top rail, the direction is changed again. Thus the alphabets of the message are written in a zig-zag manner.

After each alphabet has been written, the individual rows are combined to obtain the cipher-text.

Decryption

As we’ve seen earlier, the number of columns in rail fence cipher remains equal to the length of plain-text message. And the key corresponds to the number of rails.

Hence, rail matrix can be constructed accordingly. Once we’ve got the matrix we can figure-out the spots where texts should be placed (using the same way of moving diagonally up and down alternatively ).

Then, we fill the cipher-text row wise. After filling it, we traverse the matrix in zig-zag manner to obtain the original text.

**Program Code:**

#include <bits/stdc++.h>

using namespace std;

string encrypt(string text, int rails) {

    char railMatrix[rails][text.length()];

    for (int i = 0; i < rails; i++) {

        for (int j = 0; j < text.length(); j++) {

            railMatrix[i][j] = '.';

        }

    }

    int row = 0;

    int direction = 1; // 1 for moving down, -1 for moving up

    for (int i = 0; i < text.length(); i++) {

        railMatrix[row][i] = text[i];

        if (row == 0) {

            direction = 1;

        } else if (row == rails - 1) {

            direction = -1;

        }

        row += direction;

    }

    string encryptedText = "";

    for (int i = 0; i < rails; i++) {

        for (int j = 0; j < text.length(); j++) {

            if (railMatrix[i][j] != '.') {

                encryptedText += railMatrix[i][j];

            }

        }

    }

    return encryptedText;

}

string decrypt(string text, int rails) {

    char railMatrix[rails][text.length()];

    for (int i = 0; i < rails; i++) {

        for (int j = 0; j < text.length(); j++) {

            railMatrix[i][j] = '.';

        }

    }

    int row = 0;

    int direction = 1;

    for (int i = 0; i < text.length(); i++) {

        railMatrix[row][i] = '\*';

        if (row == 0) {

            direction = 1;

        } else if (row == rails - 1) {

            direction = -1;

        }

        row += direction;

    }

    int index = 0;

    for (int i = 0; i < rails; i++) {

        for (int j = 0; j < text.length(); j++) {

            if (railMatrix[i][j] == '\*' && index < text.length()) {

                railMatrix[i][j] = text[index++];

            }

        }

    }

    string decryptedText = "";

    row = 0;

    direction = 1;

    for (int i = 0; i < text.length(); i++) {

        decryptedText += railMatrix[row][i];

        if (row == 0) {

            direction = 1;

        } else if (row == rails - 1) {

            direction = -1;

        }

        row += direction;

    }

    return decryptedText;

}

int main() {

    string text;

    int rails;

    cout << "Enter the text to encrypt: ";

    getline(cin, text);

    cout << "Enter the number of rails: ";

    cin >> rails;

    string encryptedText = encrypt(text, rails);

    cout << "Encrypted: " << encryptedText << endl;

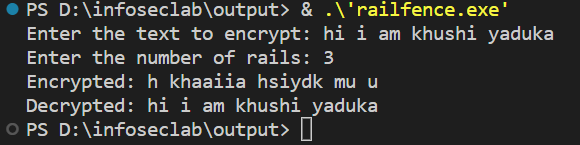
    string decryptedText = decrypt(encryptedText, rails);

    cout << "Decrypted: " << decryptedText << endl;

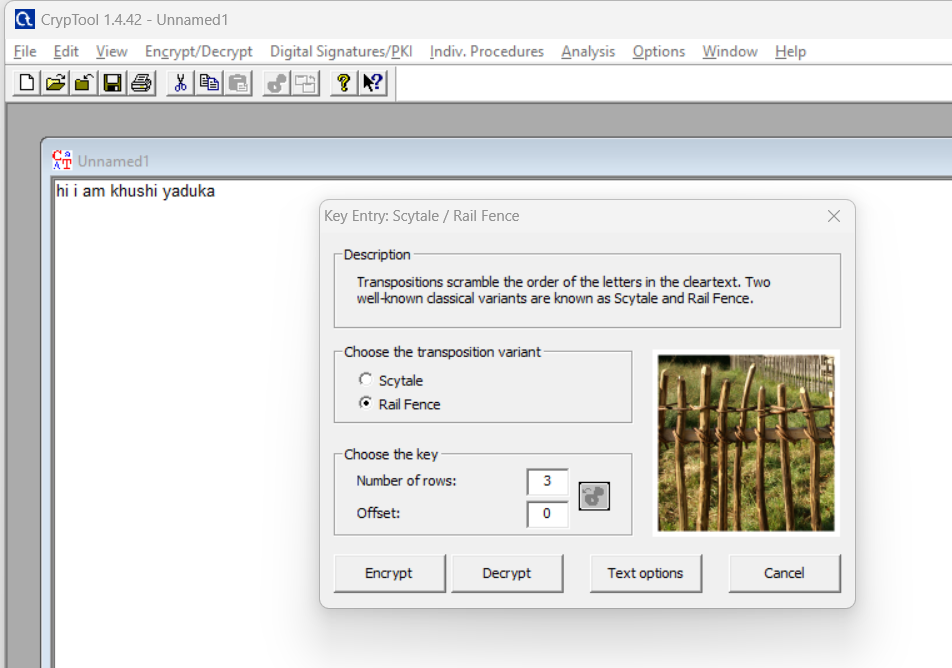
    return 0;

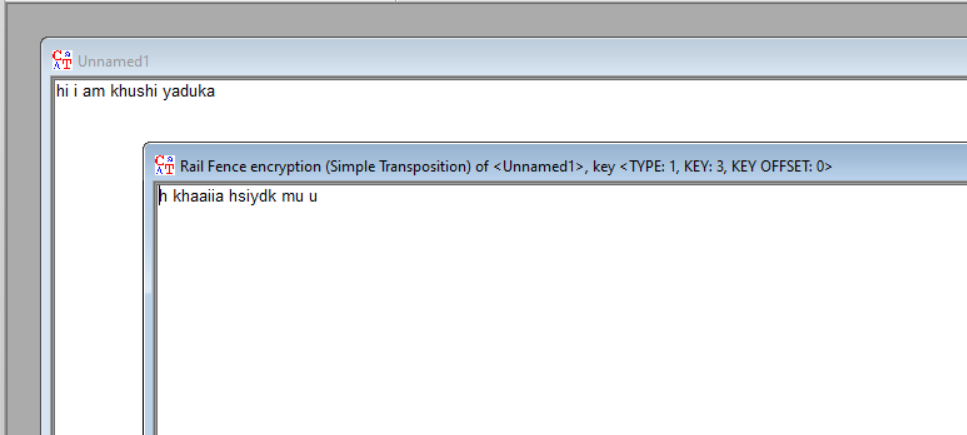
}

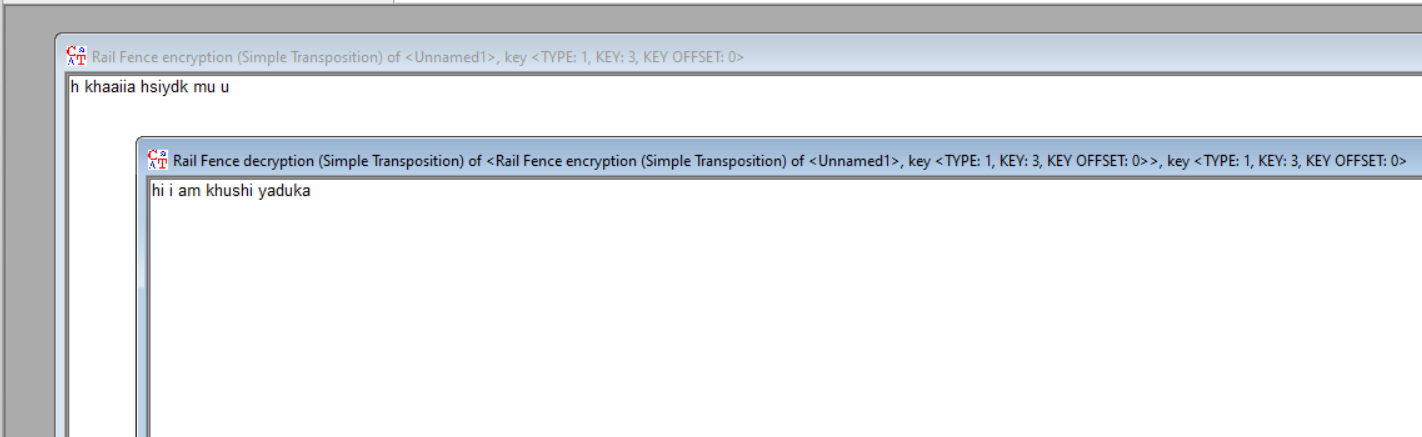
**Program Output(IDE):**

****

**Program Output (Cryptool):**

****

****

****

**Cryptanalysis:**

**Advantages of Railfence Transposition Cipher:**

**Quick Encryption**: Encrypting a message using the Rail Fence cipher is relatively fast and doesn't require complex mathematical operations or extensive computation.

**Visual Complexity**: The encrypted message has a visually complex appearance with diagonal lines that can make it appear more secure than it actually is, which might deter casual eavesdroppers.

**Variable Security**: The security of the Rail Fence cipher can be increased by using a larger number of rails, making it harder to decipher manually.

**Disadvantages of Railfence Transposition Cipher:**

**Weak Security**: The Rail Fence cipher offers weak security because it is vulnerable to various cryptographic attacks, including brute force methods. With modern computing power, it can be easily cracked.

**Low Key Space**: The key space (possible number of keys) for the Rail Fence cipher is relatively small, making it susceptible to exhaustive search attacks.

**Key Sharing Challenges**: Sharing the key securely can be difficult, as there is often no clear indication of what the correct number of rails should be.

**Time Complexity:**

**The complexity of the rail fence technique depends on the number of rows used to write out the message.** The more rows we use, the more complex the encryption will be. Additionally, increasing the number of rows can make it hard for an attacker to determine the original positions of the letters in the message. Time Complexity = O(Row\*Col).

**Applications:**

* **Pen-and-Paper Puzzles:** The Rail Fence Cipher is often used in puzzles and games to challenge participants to decrypt a message as part of a brainteaser or scavenger hunt.
* **Education:** It is used as a teaching tool to introduce students to the concept of cryptography and basic encryption techniques due to its simplicity and ease of understanding.
* **Steganography:** The Rail Fence Cipher can be used as a form of steganography, where it's used to hide messages within other documents or images. It can be a simple way to obscure a message within a larger body of text or data.
* **Recreational Cryptography:** Enthusiasts sometimes use it for recreational or hobbyist cryptography projects and challenges. It can be a fun way to create and solve puzzles.

**References:**

* GeeksforGeeks(<https://www.geeksforgeeks.org/rail-fence-cipher-encryption-decryption/>)
* https://www.baeldung.com/cs/cryptography-rail-fence-technique

**EXPERIMENT NO: 5**

**Aim:** Study and implement a program for Vigenere cipher to encrypt and decrypt the message .

**Introduction:** Vigenere Cipher is a method of encrypting alphabetic text. It uses a simple form of polyalphabetic substitution. A polyalphabetic cipher is any cipher based on substitution, using multiple substitution alphabets.The encryption of the original text is done using the Vigenère square or Vigenère table.

A more easy implementation could be to visualize Vigenère algebraically by converting [A-Z] into numbers [0–25].

**Encryption**

The plaintext(P) and key(K) are added modulo 26.

Ei = (Pi + Ki) mod 26

**Decryption**

Di = (Ei - Ki) mod 26

**Program Code:**

#include <bits/stdc++.h>

using namespace std;

string vigenereEncrypt(const string &plainText, const string &key) {

    string encryptedText = "";

    for (int i = 0, j = 0; i < plainText.length(); ++i) {

        char plainChar = plainText[i];

        char keyChar = key[j % key.length()];

        if (islower(plainChar)) {

            char base = 'a';

            char encryptedChar = (plainChar + keyChar - 2 \* base) % 26 + base;

            encryptedText += encryptedChar;

            ++j;

        } else {

            encryptedText += plainChar;

        }

    }

    return encryptedText;

}

string vigenereDecrypt(const string &encryptedText, const string &key) {

    string decryptedText = "";

    for (int i = 0, j = 0; i < encryptedText.length(); ++i) {

        char encryptedChar = encryptedText[i];

        char keyChar = key[j % key.length()];

        if (islower(encryptedChar)) {

            char base = 'a';

            char decryptedChar = (encryptedChar - keyChar + 26) % 26 + base;

            decryptedText += decryptedChar;

            ++j;

        } else {

            decryptedText += encryptedChar;

        }

    }

    return decryptedText;

}

int main() {

    string plainText, key;

    cout << "Enter the plaintext: ";

    getline(cin, plainText);

    cout << "Enter the key: ";

    getline(cin, key);

    string encryptedText = vigenereEncrypt(plainText, key);

    string decryptedText = vigenereDecrypt(encryptedText, key);

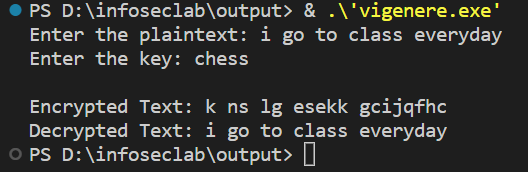
    cout << "\nEncrypted Text: " << encryptedText << endl;

    cout << "Decrypted Text: " << decryptedText << endl;

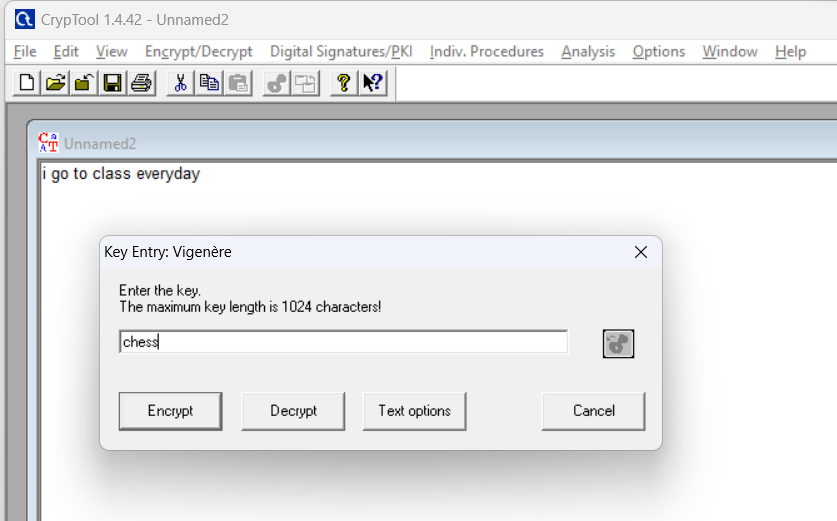
    return 0;

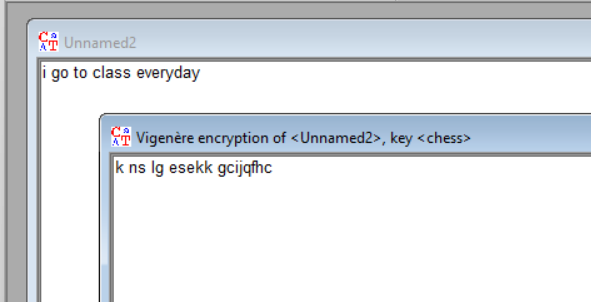
}

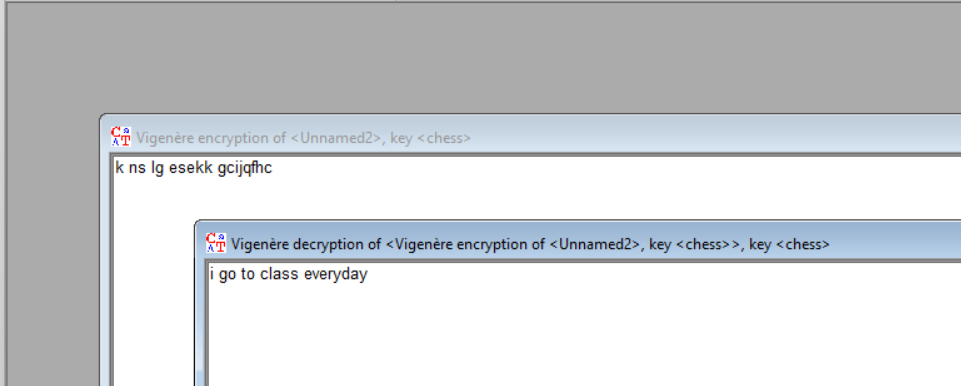
**Program Output(IDE):**

****

**Program Output (Cryptool):**

****

****

****

**Cryptanalysis:**

**Advantages of Vigenere Cipher:**

* **Polyalphabetic Substitution:** Unlike simple Caesar ciphers, which use a single fixed shift value for all characters, the Vigenère cipher uses a keyword to create different shift values for different positions in the plaintext. This makes it more complex and resistant to simple frequency analysis.
* **Variable Key Length:** The Vigenère cipher allows for the use of variable-length keywords, which can make it more adaptable and potentially more secure.
* **Historical Significance:** The Vigenère cipher has historical significance and was used for centuries, including by famous historical figures. It's interesting from a historical and educational perspective.

**Disadvantages of Vigenere Cipher :**

* **Repeating Key length:** The primary weakness of the Vigenère cipher is the repeating nature of its [key](https://en.wikipedia.org/wiki/Key_(cryptography)). If a cryptanalyst correctly guesses the key's length *n*, the cipher text can be treated as *n* interleaved [Caesar ciphers](https://en.wikipedia.org/wiki/Caesar_cipher), which can easily be broken individually. The key length may be discovered by [brute force](https://en.wikipedia.org/wiki/Brute-force_attack) testing each possible value of *n.*
* **Vulnerable to Frequency Analysis:** While the Vigenère cipher is more secure than simple Caesar ciphers, it is still vulnerable to frequency analysis. With a long enough ciphertext, attackers can still analyze patterns in the frequency of letters and potentially crack the code.
* **Weak Security for Modern Purposes:** In the context of modern cryptography, the Vigenère cipher is considered weak and insecure. It can be easily broken with modern computer-based methods, including brute force attacks or using known-plaintext attacks.

**Time Complexity:**

Time complexity of Vigenere Cipher is **O(n),** where n is the length of the string. This is because, for each character in the plaintext, you perform a simple character substitution operation based on the key. Since both the plaintext and key have the same length, you perform a constant-time operation for each character in the message.

**Applications:**

* **Educational Purposes:** The Vigenère cipher can be a useful tool for teaching the concept of polyalphabetic substitution and basic encryption techniques. It helps students understand the principles of cryptography, including key management and substitution ciphers.
* **Puzzles and Games:** The Vigenère cipher can be used to create puzzles and games. Cryptogram puzzles, for example, often use simple substitution ciphers like the Vigenère to encode messages that readers need to decipher.
* **Historical Documents Analysis:** When studying historical documents, researchers may encounter encrypted messages that were encrypted using methods like the Vigenère cipher. Deciphering these messages can provide insights into the history and context of the document.
* **Steganography:** Although not a highly secure method, the Vigenère cipher can be used in steganography, the practice of hiding messages within other media. In this context, it's used more for obfuscation than security.

**References:**

* Wikipedia(https://en.wikipedia.org/wiki/Vigen%C3%A8re\_cipher)
* GeeksforGeeks(<https://www.geeksforgeeks.org/vigenere-cipher/>)

**EXPERIMENT NO: 6**

**Aim:** Study and implement a program for 5\*5 Playfair cipher.

**Introduction:** Playfair cipher is an encryption algorithm to encrypt or encode a message. It is the same as a traditional cipher. The only difference is that it encrypts a **digraph** (a pair of two letters) instead of a single letter.

It initially creates a key-table of 5\*5 matrix. The matrix contains alphabets that act as the key for encryption of the plaintext. Note that any alphabet should not be repeated. Another point to note that there are 26 alphabets and we have only 25 blocks to put a letter inside it. Therefore, one letter is excess so, a letter will be omitted (usually J) from the matrix. Nevertheless, the plaintext contains J, then **J** is replaced by **I**. It means treat I and J as the same letter, accordingly.

Algorithm to encrypt the plain text: The plaintext is split into pairs of two letters (digraphs). If there is an odd number of letters, a Z is added to the last letter.

Pair cannot be made with same letter. Break the letter in single and add a bogus letter to the previous letter. If both the letters are in the same column: Take the letter below each one (going back to the top if at the bottom).

If both the letters are in the same row: Take the letter to the right of each one (going back to the leftmost if at the rightmost position).

If neither of the above rules is true: Form a rectangle with the two letters and take the letters on the horizontal opposite corner of the rectangle.

Decrypting the Playfair cipher is as simple as doing the same process in reverse. The receiver has the same key and can create the same key table, and then decrypt any messages made using that key

**Program Code:**

import java.awt.Point;

import java.util.Scanner;

public class PlayfairCipher {

    private int length = 0;

    private String[][] table;

    public static void main(String args[]) {

        new PlayfairCipher().start();

    }

    private void start() {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the key for Playfair cipher: ");

        String key = parseString(sc);

        while (key.equals("")) {

            System.out.println("Key cannot be empty. Please enter a valid key.");

        }

        table = this.cipherTable(key);

        System.out.print("Enter the plaintext to be encipher: ");

        String input = parseString(sc);

        while (input.equals("")) {

            System.out.println("Plaintext cannot be empty. Please enter a valid plaintext.");

        }

        String output = cipher(input);

        String decodedOutput = decode(output);

        keyTable(table);

        printResults(output, decodedOutput);

        sc.close();

    }

    private String parseString(Scanner sc) {

        String parse = sc.nextLine();

        parse = parse.toUpperCase();

        parse = parse.replaceAll("[^A-Z]", "");

        parse = parse.replace("J", "I");

        return parse;

    }

    private String[][] cipherTable(String key) {

        String[][] playfairTable = new String[5][5];

        String keyString = key + "ABCDEFGHIKLMNOPQRSTUVWXYZ";

        for (int i = 0; i < 5; i++)

            for (int j = 0; j < 5; j++)

                playfairTable[i][j] = "";

        for (int k = 0; k < keyString.length(); k++) {

            boolean repeat = false;

            boolean used = false;

            for (int i = 0; i < 5; i++) {

                for (int j = 0; j < 5; j++) {

                    if (playfairTable[i][j].equals("" + keyString.charAt(k))) {

                        repeat = true;

                    } else if (playfairTable[i][j].equals("") && !repeat && !used) {

                        playfairTable[i][j] = "" + keyString.charAt(k);

                        used = true;

                    }

                }

            }

        }

        return playfairTable;

    }

    private String cipher(String in) {

        length = (int) in.length() / 2 + in.length() % 2;

        for (int i = 0; i < (length - 1); i++) {

            if (in.charAt(2 \* i) == in.charAt(2 \* i + 1)) {

                in = new StringBuffer(in).insert(2 \* i + 1, 'X').toString();

                length = (int) in.length() / 2 + in.length() % 2;

            }

        }

        String[] digraph = new String[length];

        for (int j = 0; j < length; j++) {

            if (j == (length - 1) && in.length() / 2 == (length - 1))

                in = in + "X";

            digraph[j] = in.charAt(2 \* j) + "" + in.charAt(2 \* j + 1);

        }

        String out = "";

        String[] encDigraphs = new String[length];

        encDigraphs = encodeDigraph(digraph);

        for (int k = 0; k < length; k++)

            out = out + encDigraphs[k];

        return out;

    }

    private String[] encodeDigraph(String di[]) {

        String[] encipher = new String[length];

        for (int i = 0; i < length; i++) {

            char a = di[i].charAt(0);

            char b = di[i].charAt(1);

            int r1 = (int) getPoint(a).getX();

            int r2 = (int) getPoint(b).getX();

            int c1 = (int) getPoint(a).getY();

            int c2 = (int) getPoint(b).getY();

            if (r1 == r2) {

                c1 = (c1 + 1) % 5;

                c2 = (c2 + 1) % 5;

            } else if (c1 == c2) {

                r1 = (r1 + 1) % 5;

                r2 = (r2 + 1) % 5;

            } else {

                int temp = c1;

                c1 = c2;

                c2 = temp;

            }

            encipher[i] = table[r1][c1] + "" + table[r2][c2];

        }

        return encipher;

    }

    private String decode(String out) {

        String decoded = "";

        for (int i = 0; i < out.length() / 2; i++) {

            char a = out.charAt(2 \* i);

            char b = out.charAt(2 \* i + 1);

            int r1 = (int) getPoint(a).getX();

            int r2 = (int) getPoint(b).getX();

            int c1 = (int) getPoint(a).getY();

            int c2 = (int) getPoint(b).getY();

            if (r1 == r2) {

                c1 = (c1 + 4) % 5;

                c2 = (c2 + 4) % 5;

            } else if (c1 == c2) {

                r1 = (r1 + 4) % 5;

                r2 = (r2 + 4) % 5;

            } else {

                int temp = c1;

                c1 = c2;

                c2 = temp;

            }

            decoded = decoded + table[r1][c1] + table[r2][c2];

        }

        return decoded;

    }

    private Point getPoint(char c) {

        Point pt = new Point(0, 0);

        for (int i = 0; i < 5; i++)

            for (int j = 0; j < 5; j++)

                if (c == table[i][j].charAt(0))

                    pt = new Point(i, j);

        return pt;

    }

    private void keyTable(String[][] printTable) {

        System.out.println("Playfair Cipher Key Matrix:");

        System.out.println();

        for (int i = 0; i < 5; i++) {

            for (int j = 0; j < 5; j++) {

                System.out.print(printTable[i][j] + " ");

            }

            System.out.println();

        }

        System.out.println();

    }

    private void printResults(String encipher, String dec) {

        System.out.print("Encrypted Message: ");

        System.out.println(encipher);

        System.out.println();

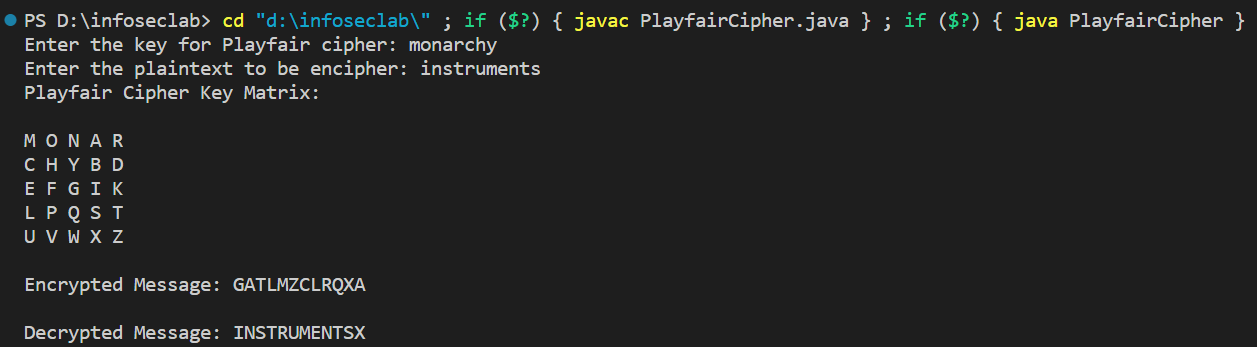
        System.out.print("Decrypted Message: ");

        System.out.println(dec);

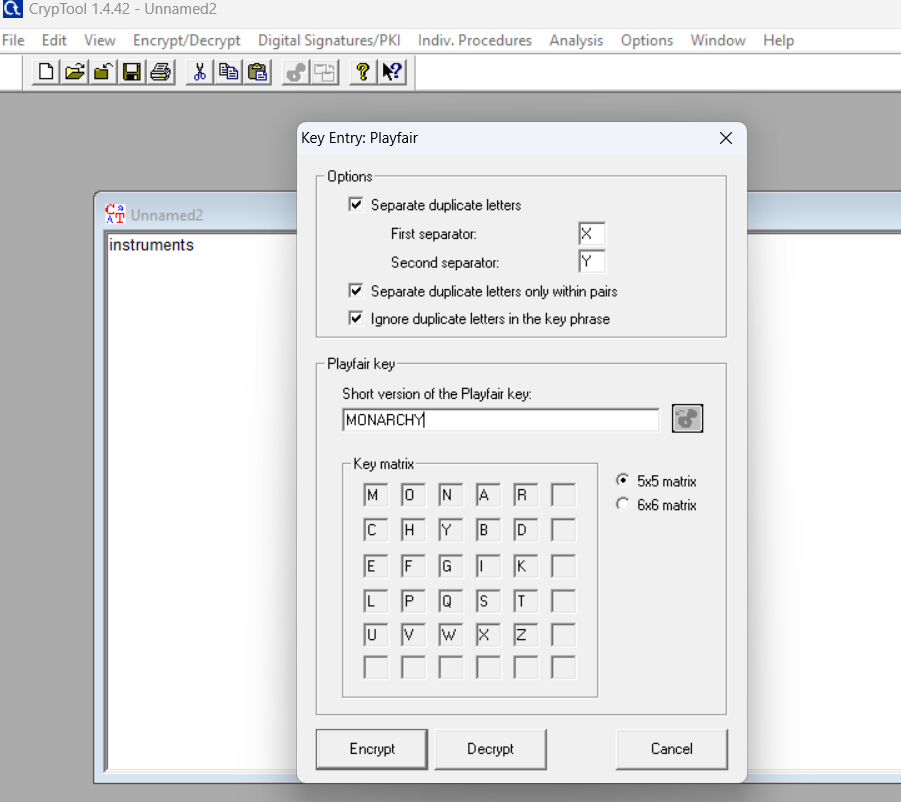
    }

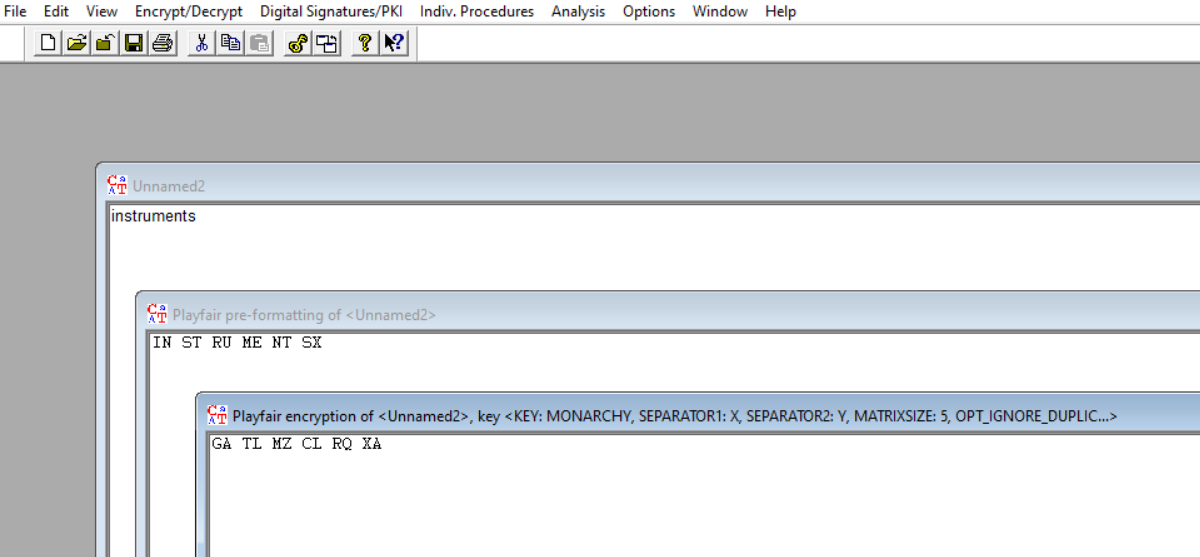
}

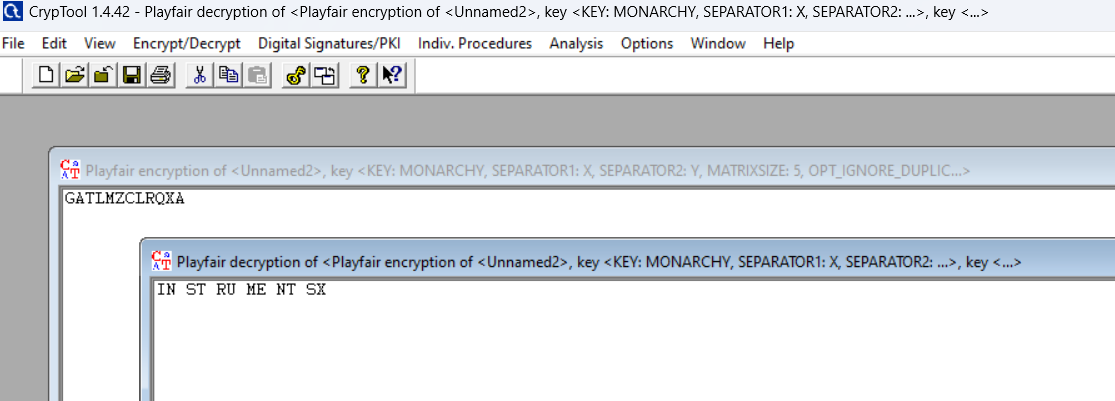
**Program Output(IDE):**

****

**Program Output (Cryptool):**

****

****

****

**Cryptanalysis:**

**Advantages of Playfair Cipher:**

* **Polygram Substitution:** Encrypts pairs of letters together, increasing complexity.
* **Enhanced Security:** Offers more security compared to simple substitution ciphers with moderate complexity.
* **Key Management:** The 5x5 matrix provides a clear visual representation of the key.
* **Suitable for Pen-and-Paper:** Can be implemented manually without special tools.

**Disadvantages of Playfair Cipher:**

* **Limited Character Set:** Cannot encrypt numbers, symbols, or non-alphabetic characters.
* **Lack of Perfect Secrecy:** Doesn't achieve perfect secrecy, making it less secure.
* **Unsuitable for Modern Cryptography:** Not recommended for highly sensitive data.

**Time Complexity:**

**Encryption:** The time complexity for encrypting a message using the Playfair cipher is typically O(n), where n is the length of the message. This is because you need to process each pair of letters in the message, look up their positions in the key matrix, and apply the encryption rules.

**Decryption:** Decryption using the Playfair cipher also has a time complexity of O(n), as you need to process each pair of letters in the ciphertext, find their positions in the key matrix, and apply the decryption rules.

**Applications:**

* Applications of the Playfair cipher include military communications, cryptography puzzles, and cryptanalysis studies.
* It was used for tactical purposes by British forces in the Second Boer War and in World War I and for the same purpose by the Australians during World War II.
* It has also been used in digital encryption systems, such as the CAST-128 and A5/1 encryption algorithms.

**References:**

* Wikipedia()
* GeeksforGeeks(https://www.geeksforgeeks.org/playfair-cipher-with-examples/)

**EXPERIMENT NO: 7**

**Aim:** Study and implement a program for n-gram Hill cipher.

**Introduction:**

Hill cipher is a polygraphic substitution cipher based on linear algebra.Each letter is represented by a number modulo 26. Often the simple scheme A = 0, B = 1, …, Z = 25 is used, but this is not an essential feature of the cipher. To encrypt a message, each block of n letters (considered as an n-component vector) is multiplied by an invertible n × n matrix, against modulus 26. To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption.

The matrix used for encryption is the cipher key, and it should be chosen randomly from the set of invertible n × n matrices (modulo 26).

Decryption

To decrypt the message, we turn the ciphertext back into a vector, then simply multiply by the inverse matrix of the key matrix.

**Program Code:**

import java.util.Scanner;

public class Hill {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);

        System.out.println("|Enter 1 to Encrypt and 0 to Decrypt|");

        System.out.print("Enter a number : ");

        int choice = in.nextInt();

        in.nextLine();

        if(choice==1){

            System.out.print("Enter String to encrypt : ");

            String str = in.nextLine();

            System.out.print("enter key: ");

            String key = in.nextLine();

            System.out.print("String after encryption : ");

            System.out.println(encrypt(key.toLowerCase(),str.toLowerCase()));

        }

        else if(choice==0){

            System.out.println("enter String to decrypt: ");

            String str = in.nextLine();

            System.out.print("enter key: ");

            String key = in.nextLine();

            decrypt(str.toLowerCase(),key.toLowerCase());

        }

        else{

            System.out.println("enter a valid choice!");

        }

        System.out.println("=========================================================");

    }

    public static String encrypt(String key,String plainText){

        if(!isPerfectSquare(key.length())){

            return "enter a key whose length is a perfect square root.";

        }

        String ans = "";

        int root = (int)Math.sqrt(key.length());

        int k = 0;

        int[][] keyMat = new int[root][root];

        for (int i = 0; i < root; i++) {

            for (int j = 0; j < root; j++) {

                keyMat[i][j] = (key.charAt(k)-'a') % 26;

                k++;

            }

        }

        int lengthPlain = plainText.length();

        int[][] plainMat;

        if(root>=lengthPlain){

            for (int i = 0; i < root-lengthPlain; i++) {

                plainText += 'x';

                lengthPlain++;

            }

            plainMat = new int[root][1];

        } else{

            while(lengthPlain%root!=0){

                plainText += 'x';

                lengthPlain++;

            }

            plainMat = new int[root][lengthPlain/root];

        }

        int temp = 0;

        for (int i = 0; i < lengthPlain/root; i++) {

            for (int j = 0; j < root; j++) {

                plainMat[j][i] = (plainText.charAt(temp) -'a') % 26;

                temp++;

            }

        }

        char[][] finalMat = matrixMul(keyMat,plainMat);

        for (int i = 0; i < finalMat[0].length; i++) {

            for (int j = 0; j < finalMat.length; j++) {

                ans += finalMat[j][i];

            }

        }

//        for (int i = 0; i < root; i++) {

//            System.out.println(Arrays.toString(keyMat[i]));

//        }

//        System.out.println("=============================");

//        for (int i = 0; i < root; i++) {

//            System.out.println(Arrays.toString(plainMat[i]));

//        }

        return ans;

    }

    public static char[][] matrixMul(int[][] keyMat, int[][] plainMat) {

        char finalMat[][] = new char[keyMat.length][plainMat[0].length];

        for (int i = 0; i < finalMat.length; i++) {

            for (int j = 0; j < finalMat[i].length; j++) {

                for (int k = 0; k < keyMat[0].length; k++) {

                    finalMat[i][j] += (keyMat[i][k] \* plainMat[k][j]) ;

                }

                finalMat[i][j] = (char)((finalMat[i][j] %26) + 97) ;

            }

        }

//        for (int i = 0; i < finalMat.length; i++) {

//            System.out.println(Arrays.toString(finalMat[i]));

//        }

//        System.out.println("==============================");

        return finalMat;

    }

    public static boolean isPerfectSquare(int a){

        int root = (int)Math.sqrt(a);

        return root\*root == a;

    }

    public static int[][] calculateAdjoint(int[][] matrix) {

        int size = matrix.length;

        int[][] adjoint = new int[size][size];

        if (size == 1) {

            adjoint[0][0] = 1;

        } else if (size == 2) {

            adjoint[0][0] = matrix[1][1];

            adjoint[0][1] = -matrix[0][1];

            adjoint[1][0] = -matrix[1][0];

            adjoint[1][1] = matrix[0][0];

        } else {

            for (int i = 0; i < size; i++) {

                for (int j = 0; j < size; j++) {

                    int[][] submatrix = getSubmatrix(matrix, i, j);

                    adjoint[i][j] = (int) Math.pow(-1, i + j) \* calculateDeterminant(submatrix);

                }

            }

            adjoint = transposeMatrix(adjoint);

        }

        return adjoint;

    }

    public static int[][] getSubmatrix(int[][] matrix, int rowToRemove, int colToRemove) {

        int size = matrix.length;

        int[][] submatrix = new int[size - 1][size - 1];

        int rowIndex = 0;

        int colIndex;

        for (int i = 0; i < size; i++) {

            if (i == rowToRemove) {

                continue;

            }

            colIndex = 0;

            for (int j = 0; j < size; j++) {

                if (j == colToRemove) {

                    continue;

                }

                submatrix[rowIndex][colIndex] = matrix[i][j];

                colIndex++;

            }

            rowIndex++;

        }

        return submatrix;

    }

    public static int calculateDeterminant(int[][] matrix) {

        int size = matrix.length;

        if (size == 2) {

            return matrix[0][0] \* matrix[1][1] - matrix[0][1] \* matrix[1][0];

        }

        int determinant = 0;

        for (int i = 0; i < size; i++) {

            determinant += matrix[0][i] \* (int) Math.pow(-1, i) \* calculateDeterminant(getSubmatrix(matrix, 0, i));

        }

        return determinant;

    }

    public static int[][] transposeMatrix(int[][] matrix) {

        int rows = matrix.length;

        int cols = matrix[0].length;

        int[][] transposed = new int[cols][rows];

        for (int i = 0; i < rows; i++) {

            for (int j = 0; j < cols; j++) {

                transposed[j][i] = matrix[i][j];

            }

        }

        return transposed;

    }

    /\*--------------------------------------------------------------------------------\*/

    // finding the modulo inverse of number A under modulo 'M'

    public static int modInverse(int A, int M) {

        for (int i = 1; i < M; i++) {

            if (((A % M) \* (i % M)) % M == 1) {

                return i;

            }

        }

        return -1;

    }

    public static int[][] inverseOfKeyMatrix(int[][] keyMatrix) {

        int determinant = calculateDeterminant(keyMatrix);

        if (determinant < 0) {

            determinant = 26 - (Math.abs(determinant) % 26);

        }

        determinant = determinant % 26;

        int modularInverse = modInverse(determinant, 26);

        int[][] inverseMatrix = calculateAdjoint(keyMatrix);

        for (int i = 0; i < inverseMatrix.length; i++) {

            for (int j = 0; j < inverseMatrix[0].length; j++) {

                inverseMatrix[i][j] = (inverseMatrix[i][j]) % 26;

                if (inverseMatrix[i][j] < 0) {

                    inverseMatrix[i][j] = 26 - Math.abs(inverseMatrix[i][j]);

                }

                inverseMatrix[i][j] = (inverseMatrix[i][j] \* modularInverse) % 26;

            }

        }

        return inverseMatrix;

    }

    public static void decrypt(String str,String key){

        if(!isPerfectSquare(key.length())){

            System.out.println("enter a key whose length is a perfect square root.");

        }

        int root = (int)Math.sqrt(key.length());

        int[][] keyMat = new int[root][root];

        int k=0;

        for (int i = 0; i < root; i++) {

            for (int j = 0; j < root; j++) {

                keyMat[i][j] = (key.charAt(k)-'a') % 26;

                k++;

            }

        }

        int temp =0;

        int[][] inverseKey = inverseOfKeyMatrix(keyMat);

        int[][] strMat = new int[root][str.length()/root];

        for (int i = 0; i < strMat[0].length; i++) {

            for (int j = 0; j < strMat.length; j++) {

                strMat[j][i] = (str.charAt(temp)-'a') % 26;

                temp++;

            }

        }

        char[][] finalText = matrixMul(inverseKey,strMat);

        String ans = "";

        for (int i = 0; i < finalText[0].length; i++) {

            for (int j = 0; j < finalText.length; j++) {

                ans += finalText[j][i];

            }

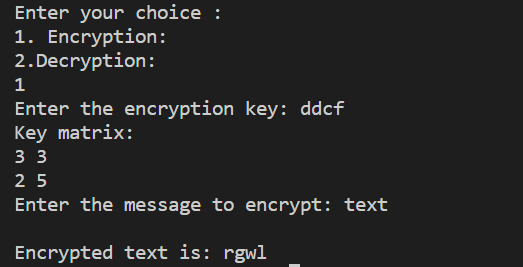
        }

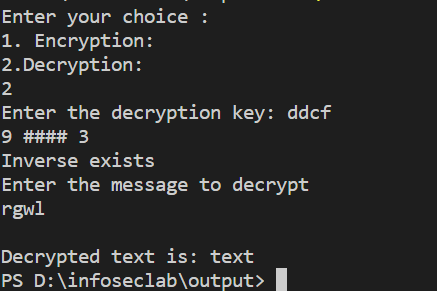
        System.out.println("Plaintext : " + ans);

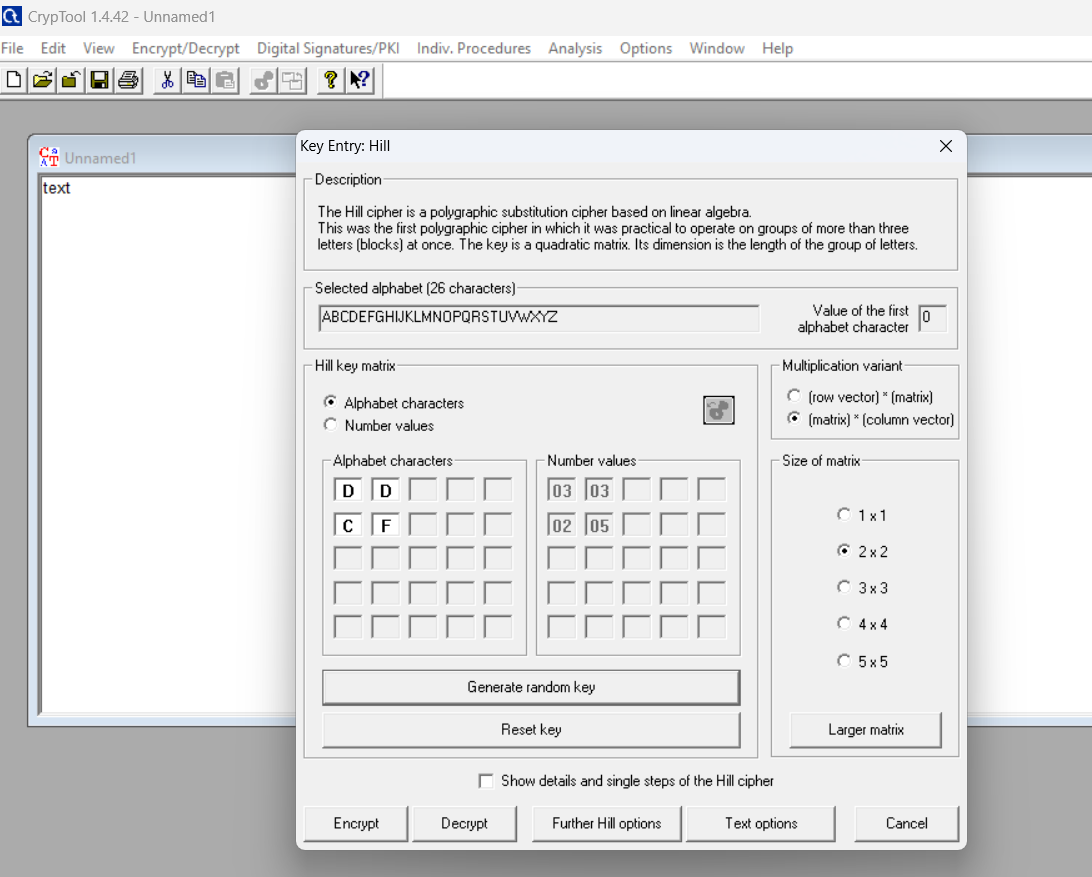
    }

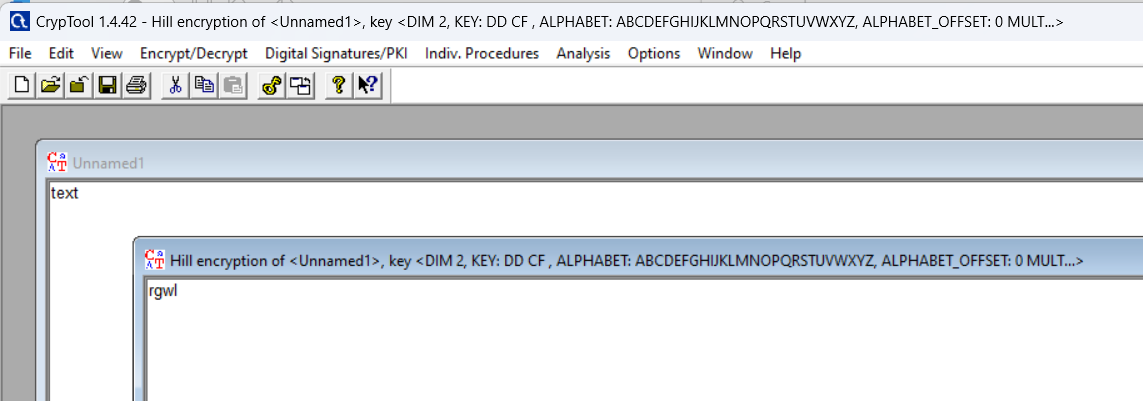
}

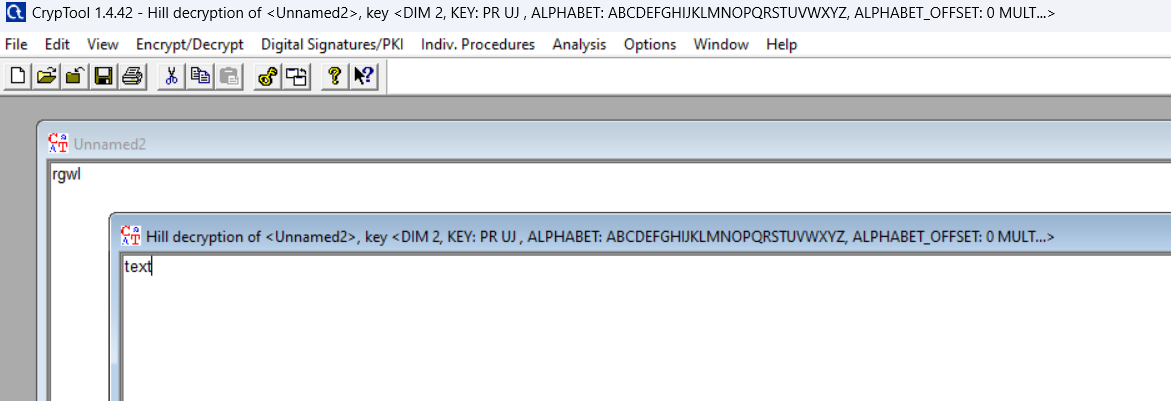
**Program Output(IDE):**

****

****

**Program Output (Cryptool):**

****

****

**Cryptanalysis:**

**Key length:** An important factor in the security of a Hill cipher is the size of its key matrix. In a Hill Cipher with a key matrix of size n x n, with a dimension of dimension n, the key space grows exponentially with n. Consequently, larger key matrices are generally more secure, but they also increase computational complexity.

**Known plaintext attack:** If the attacker has access to the plaintext pairs and associated ciphertext, a known plaintext attack can be used to recover the key. The complexity of this attack depends on the size of the key matrix.

**Time Complexity:**

The time complexity in Hill cipher encryption and decryption is O(n^3), where n is the dimension of the key matrix.

**Applications:**

* **Image Encryption:**While not as common as other encryption techniques, Hill cipher can be adapted for encrypting images. Each pixel's color values can be treated as numerical values and encrypted using the Hill cipher.
* **Securing Text:**The primary application of the Hill Cipher is in securing text messages. It uses a matrix as a key to encrypt and decrypt messages, making it a useful tool for ensuring the confidentiality of sensitive information transmitted over open networks.
* **Multimedia security:** Hill Cipher is employed in multimedia security as an inexpensive and robust tool. It is particularly useful in protecting multimedia content from unauthorized access or tampering due to its simplicity and efficiency.

**References:**

* CodingNinjas (https://www.codingninjas.com/studio/library/polyalphabetic-hill-cipher)
* GeeksforGeeks(https://www.geeksforgeeks.org/hill-cipher/)

**EXPERIMENT 8**

**Aim:** Use Crypto++ library to implement encryption and decryption of different block ciphers.

**Introduction:**

The field of cryptography is vital for ensuring the security and confidentiality of data in various applications, such as secure communication, data storage, and authentication. Cryptographic algorithms play a crucial role in protecting sensitive information from unauthorized access. In this experiment, we explore the implementation of encryption and decryption using the Crypto++ library, a popular C++ library for cryptographic operations. Specifically, we focus on various block ciphers, which are symmetric key algorithms used for data encryption and decryption.

**Block Ciphers**

Block ciphers are a class of symmetric key ciphers that operate on fixed-size blocks of data, typically 128, 192, or 256 bits. These ciphers use the same key for both encryption and decryption, making them well-suited for applications where data needs to be secured with a shared secret key. Some widely used block ciphers include the Advanced Encryption Standard (AES), Data Encryption Standard (DES), and Triple DES (3DES). Each of these ciphers employs distinct encryption and decryption algorithms, making them suitable for different use cases.

**The Crypto++ Library**

Crypto++ is a powerful and versatile C++ library that provides implementations of various cryptographic algorithms, including block ciphers. It offers a standardized interface for encryption and decryption operations, making it a valuable tool for secure data processing. The library includes classes and functions for AES, DES, 3DES, and many other encryption algorithms, allowing developers to integrate cryptographic functionality into their applications seamlessly.

**AES Encryption and Decryption**

The Advanced Encryption Standard (AES) is one of the most widely used block ciphers. AES operates on fixed 128-bit blocks of data and supports key sizes of 128, 192, and 256 bits. It uses a substitution-permutation network (SPN) structure, which includes substitution, permutation, and key mixing layers to provide robust encryption. In our experiment, we will demonstrate how to implement AES encryption and decryption using the Crypto++ library. AES is known for its security, efficiency, and wide adoption in various applications.

DES and 3DES Encryption and Decryption

The Data Encryption Standard (DES) and Triple DES (3DES) are older block ciphers that have been widely used in the past. DES operates on 64-bit blocks and uses a 56-bit key. 3DES is an enhancement of DES and provides increased security by applying the DES algorithm three times in succession. In our experiment, we will explore how to implement both DES and 3DES encryption and decryption using Crypto++. While these ciphers are considered legacy due to their smaller key sizes, they are still relevant in some applications.

**CODE:**

#include <cryptopp/modes.h>

#include <cryptopp/aes.h>

#include <cryptopp/des.h>

#include <cryptopp/filters.h>

#include <cryptopp/hex.h>

#include <iostream>

using namespace CryptoPP;

int main() {

std::string plaintext = "Hello, World!";

std::string ciphertext;

std::string decryptedtext;

// AES

{

byte key[AES::DEFAULT\_KEYLENGTH];

memset(key, 0x00, AES::DEFAULT\_KEYLENGTH);

ECB\_Mode< AES >::Encryption e;

e.SetKey(key, AES::DEFAULT\_KEYLENGTH);

StringSource ss1(plaintext, true,

new StreamTransformationFilter(e,

new StringSink(ciphertext)

)

);

ECB\_Mode< AES >::Decryption d;

d.SetKey(key, AES::DEFAULT\_KEYLENGTH);

StringSource ss2(ciphertext, true,

new StreamTransformationFilter(d,

new StringSink(decryptedtext)

)

);

std::cout << "AES ciphertext: " << ciphertext << std::endl;

std::cout << "AES decryptedtext: " << decryptedtext << std::endl;

}

ciphertext.clear();

decryptedtext.clear();

// DES

{

byte key[DES\_EDE2::DEFAULT\_KEYLENGTH];

memset(key, 0x00, DES\_EDE2::DEFAULT\_KEYLENGTH);

ECB\_Mode< DES\_EDE2 >::Encryption e;

e.SetKey(key, DES\_EDE2::DEFAULT\_KEYLENGTH);

StringSource ss1(plaintext, true,

new StreamTransformationFilter(e,

new StringSink(ciphertext)

)

);

ECB\_Mode< DES\_EDE2 >::Decryption d;

d.SetKey(key, DES\_EDE2::DEFAULT\_KEYLENGTH);

StringSource ss2(ciphertext, true,

new StreamTransformationFilter(d,

new StringSink(decryptedtext)

)

);

std::cout << "DES ciphertext: " << ciphertext << std::endl;

std::cout << "DES decryptedtext: " << decryptedtext << std::endl;

}

return 0;

}

**OUTPUT**

A screenshot of a computer

Description automatically generated

**Cryptanalysis :**

Cryptanalysis is the science of studying and analyzing cryptographic systems to identify vulnerabilities and weaknesses. In the context of block ciphers like AES, DES, and 3DES, cryptanalysis plays a crucial role in assessing their security and ensuring that they provide the level of protection required for various applications. Below are some aspects of cryptanalysis to consider:

1. **Brute Force Attacks:** One common form of cryptanalysis is a brute force attack, where an adversary tries all possible keys to decrypt a ciphertext. The security of a block cipher depends on the key length, as longer keys increase the computational effort required for a successful brute force attack.
2. **Known-Plaintext and Chosen-Plaintext Attacks:** Cryptanalysts often use known-plaintext and chosen-plaintext attacks to exploit vulnerabilities in block ciphers. Known-plaintext attacks are carried out with knowledge of the plaintext and corresponding ciphertext, while chosen-plaintext attacks involve choosing plaintexts for encryption.
3. **Differential and Linear Cryptanalysis:** These are sophisticated techniques used to analyze the behavior of block ciphers in relation to plaintext and ciphertext differences. They can reveal patterns and biases in the cipher's operation, which could be exploited to break the encryption.
4. **Cryptanalysis of Key Scheduling:** Weaknesses in the key scheduling algorithms of block ciphers can be targets for cryptanalysis. A successful attack on the key schedule can lead to a complete compromise of the encryption.
5. **Side-Channel Attacks**: Cryptanalysis also encompasses side-channel attacks, which exploit information leaked during the encryption process, such as power consumption or execution time. Implementations of block ciphers must be resistant to such attacks.
6. **Block Cipher Modes of Operation:** Cryptanalysis can focus on the modes of operation used with block ciphers, such as ECB, CBC, CFB, and OFB. Understanding their security properties and potential weaknesses is essential.

**Applications**

Block ciphers, including AES, DES, and 3DES, find applications in various domains due to their ability to secure data and communications. Some common applications include:

1. **Data Encryption:** Block ciphers are used to encrypt sensitive data at rest or in transit, ensuring confidentiality and preventing unauthorized access. This includes encrypting files, databases, and email communications.
2. **Network Security:** Block ciphers are integral to secure network communication, such as SSL/TLS for securing web traffic, IPsec for VPNs, and SSH for secure remote access.
3. **Wireless Security:** Wireless protocols like WPA and WPA2 use block ciphers to protect Wi-Fi networks from eavesdropping and unauthorized access.
4. **Secure Messaging:** Messaging apps often employ block ciphers to encrypt text, voice, and video messages to safeguard user privacy.
5. **Secure Storage:** Block ciphers are used to encrypt data stored on devices, such as smartphones, ensuring that data remains confidential even if the device is lost or stolen.
6. **Financial Transactions:** Encryption of financial data is vital for secure online banking and payment processing, where block ciphers help protect sensitive information.

**References:**

<https://cryptopp.com>

**Experiment 9**

**Aim:** To Study and Implement a program for RSA Algorithm to encrypt and decrypt the message.

**Introduction:**

The RSA (Rivest-Shamir-Adleman) algorithm is a widely used public-key cryptosystem that was invented by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977. RSA is renowned for its ability to securely encrypt and digitally sign data, making it a fundamental cornerstone of modern information security. The algorithm relies on the mathematical properties of large prime numbers and modular arithmetic to provide secure communication and data protection. RSA has applications in secure email, e-commerce, digital signatures, and more.

In RSA, each user has a pair of keys: a public key, which is known to everyone, and a private key, which is kept secret. The public key is used for encryption, while the private key is used for decryption. The security of RSA is grounded in the difficulty of factoring the product of two large prime numbers, which is an essential component of the algorithm. As prime factorization is a computationally intensive task, RSA encryption remains secure as long as sufficiently large keys are used.

One of the key advantages of RSA is its versatility in providing both confidentiality and authenticity in digital communication. It ensures that only the intended recipient can decrypt the message with their private key, while the sender can digitally sign their messages with their private key to prove their identity. RSA's enduring popularity, based on the principles of asymmetric cryptography, has made it a vital tool in the protection of sensitive information and the establishment of secure online transactions in our increasingly digital world.

**CODE:**

import random

# Helper function to check if a number is prime

def is\_prime(num):

    if num <= 1:

        return False

    if num <= 3:

        return True

    if num % 2 == 0 or num % 3 == 0:

        return False

    i = 5

    while i \* i <= num:

        if num % i == 0 or num % (i + 2) == 0:

            return False

        i += 6

    return True

# Helper function to compute the greatest common divisor (GCD)

def gcd(a, b):

    while b:

        a, b = b, a % b

    return a

# Helper function to find the modular multiplicative inverse

def mod\_inverse(a, m):

    m0, x0, x1 = m, 0, 1

    while a > 1:

        q = a // m

        m, a = a % m, m

        x0, x1 = x1 - q \* x0, x0

    return x1 + m0 if x1 < 0 else x1

def list\_prime(num):

    prime\_list = []

    for i in range(2, num):

        if is\_prime(i):

            prime\_list.append(i)

    return prime\_list

# Generate two large prime numbers

def generate\_prime(bits):

    num = random.getrandbits(bits)

    return prime\_number\_list[num % len(prime\_number\_list)]

# Generate public and private keys

def generate\_keys(bits):

    p = generate\_prime(bits)

    print(f"p={p}")

    q = generate\_prime(bits)

    print(f"q={q}")

    n = p \* q

    phi = (p - 1) \* (q - 1)

    e = 65537  # A common choice for the public exponent

    d = mod\_inverse(e, phi)

    return (e, n), (d, n)

# Encrypt a message

def encrypt(public\_key, message):

    e, n = public\_key

    encrypted = [pow(ord(char), e, n) for char in message]

    return encrypted

# Decrypt a message

def decrypt(private\_key, encrypted):

    d, n = private\_key

    decrypted = [chr(pow(char, d, n)) for char in encrypted]

    return ''.join(decrypted)

if \_\_name\_\_ == '\_\_main\_\_':

    prime\_number\_list = list\_prime(10000)

    public\_key, private\_key = generate\_keys(1024)

    message = "Kem Cho, Majama"

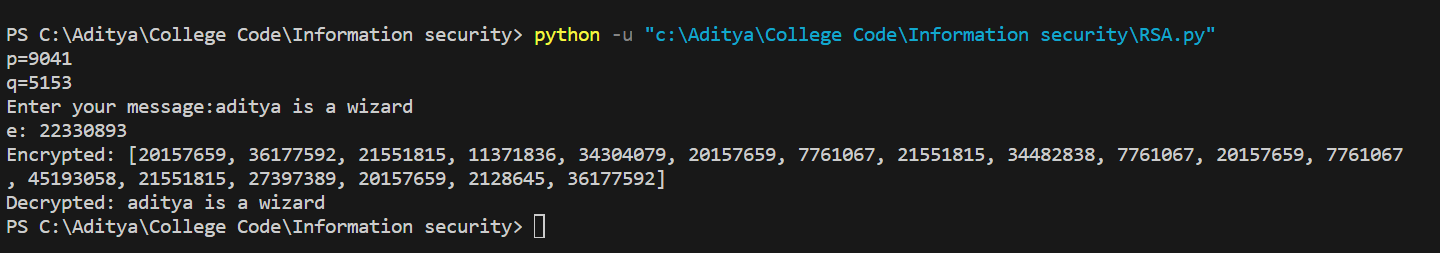
    encrypted\_message = encrypt(public\_key, message)

    print("Encrypted:", encrypted\_message)

    decrypted\_message = decrypt(private\_key, encrypted\_message)

    print("Decrypted:", decrypted\_message)

**OUTPUT(PROGRAM):**



**OUTPUT(CRYPTOOL):**

**A screenshot of a computer

Description automatically generated**

**CRYPTANALYSIS:**

Cryptanalysis of the RSA algorithm involves attempting to break its security by exploiting weaknesses in the underlying mathematics or implementation. While RSA is considered a robust encryption method, there are several cryptanalysis techniques and potential vulnerabilities to consider:

**Integer Factorization:** RSA's security relies on the difficulty of factoring the product of two large prime numbers. Cryptanalysts continually work on more efficient factoring algorithms. The development of powerful quantum computers poses a potential threat to RSA, as they may be able to factor large numbers much faster than classical computers.

**Timing Attacks**: Cryptanalysts can exploit information leakage through the timing of encryption or decryption operations. By analyzing the time taken to execute these operations, attackers might gain insights into the private key.

**Chosen Ciphertext Attacks (CCA):** In CCA attacks, an attacker can interact with an oracle to decrypt chosen ciphertexts. While modern RSA implementations incorporate padding schemes like OAEP or PKCS#1 v1.5 to mitigate these attacks, vulnerabilities may still arise if these schemes are not correctly implemented.

**Side-Channel Attacks**: RSA implementations can be susceptible to side-channel attacks based on the electromagnetic emissions, power consumption, or other observable aspects of the hardware or software during cryptographic operations. Attackers can use these observable signals to deduce private key information.

**Weak Key Generation:** Weak key generation can lead to vulnerabilities in RSA. If users generate keys with insufficient randomness or use predictable prime numbers, it becomes easier for cryptanalysts to crack the encryption. Secure key generation practices are crucial to RSA's strength.

**APPLICATIONS:**

**Secure Communication:** RSA is widely used in securing communication over the internet, particularly for encrypting email and instant messaging. It ensures that data exchanged between parties remains confidential and cannot be intercepted or deciphered by unauthorized individuals.

**Digital Signatures**: RSA is employed for creating digital signatures to verify the authenticity and integrity of digital documents, software, or messages. By signing data with a private key, a sender can prove that the content has not been tampered with and is genuinely from them.

**Secure Web Browsing:** RSA is a fundamental component of secure web browsing. It is used to establish SSL/TLS (Secure Sockets Layer/Transport Layer Security) connections, encrypting data exchanged between web browsers and servers. This ensures the privacy of sensitive information like credit card details during online transactions.

**Authentication and Access Control:** RSA is employed for user authentication in various systems, including VPNs, remote access, and secure login processes. Users have their private keys to prove their identity, which enhances security in access control.

**REFERENCES:**

Stallings, W. (2017). *Cryptography and Network Security: Principles and Practice.* Pearson. <https://www.pearson.com/en-us/subject-catalog/p/cryptography-and-network-security-principles-and-practice/P200000003477>

**Experiment No : 10**

**Aim:** To Study and Implement a program for Python code to use RSA for generation and verification of digital signature on file. It should include models for sender's authentication only and for message confidentiality and sender's authentication. Also, include a user interface and a menu.

**Introduction :**

RSA, the renowned encryption and digital signature algorithm developed by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977, plays a pivotal role in securing digital communications. This cryptographic powerhouse serves as a cornerstone in the realm of digital signatures, enabling the generation and verification of digital signatures on files. A digital signature is a cryptographic mechanism that ensures the authenticity and integrity of digital documents. In this context, RSA provides a robust solution to achieve sender authentication while also facilitating message confidentiality and sender authentication. This multifaceted functionality is crucial for a wide array of applications, from secure email communications to legal documents and financial transactions.

In the realm of digital signatures, RSA operates in two principal models: one for sender authentication only and another for both message confidentiality and sender authentication. In the sender authentication model, RSA is used to create a digital signature that can be appended to a file or document. The sender computes the signature using their private key, and recipients can verify the signature using the sender's public key. This process ensures that the document indeed originated from the claimed sender, providing a robust mechanism for sender authentication.

In the more comprehensive model, RSA not only offers sender authentication but also provides message confidentiality. In this case, the file is not only digitally signed but also encrypted using the recipient's public key. As a result, not only is the sender's authenticity guaranteed, but the contents of the message remain confidential, as only the recipient with the corresponding private key can decrypt it.

To implement RSA-based digital signatures, a user-friendly interface is essential. The user interface typically consists of a menu where users can select various options, such as "Sign File" to generate digital signatures and "Verify Signature" to confirm the authenticity and integrity of received files. This menu-driven approach simplifies the process, making it accessible to users with varying levels of technical expertise.

In essence, RSA's application in the generation and verification of digital signatures on files serves as a robust mechanism for enhancing trust and security in the digital realm. It empowers individuals and organizations to ensure that documents are tamper-proof and indeed come from the claimed sender. Moreover, when combined with encryption, it extends its utility to preserving message confidentiality. Through a user-friendly interface and menu, RSA's capabilities are harnessed effectively, making it a fundamental tool in securing modern digital communications and transactions.

**CODE:**

import os

import hashlib

from cryptography.hazmat.primitives import serialization

from cryptography.hazmat.primitives.asymmetric import rsa

from cryptography.hazmat.primitives.asymmetric import padding

from cryptography.hazmat.primitives import hashes

def generate\_key\_pair():

    private\_key = rsa.generate\_private\_key(

        public\_exponent=65537,

        key\_size=2048

    )

    private\_pem = private\_key.private\_bytes(

        encoding=serialization.Encoding.PEM,

        format=serialization.PrivateFormat.TraditionalOpenSSL,

        encryption\_algorithm=serialization.NoEncryption()

    )

    public\_key = private\_key.public\_key().public\_bytes(

        encoding=serialization.Encoding.PEM,

        format=serialization.PublicFormat.SubjectPublicKeyInfo

    )

    return private\_pem, public\_key, private\_key

def sign\_message(private\_key, message):

    private\_key = serialization.load\_pem\_private\_key(private\_key, password=None)

    signature = private\_key.sign(

        message.encode('utf-8'),

        padding.PSS(

            mgf=padding.MGF1(hashes.SHA256()),

            salt\_length=padding.PSS.MAX\_LENGTH

        ),

        hashes.SHA256()

    )

    return signature

def verify\_signature(public\_key, message, signature):

    public\_key = serialization.load\_pem\_public\_key(public\_key)

    try:

        public\_key.verify(

            signature,

            message.encode('utf-8'),

            padding.PSS(

                mgf=padding.MGF1(hashes.SHA256()),

                salt\_length=padding.PSS.MAX\_LENGTH

            ),

            hashes.SHA256()

        )

        return True

    except Exception:

        return False

def encrypt\_file(public\_key, input\_file, output\_file):

    public\_key = serialization.load\_pem\_public\_key(public\_key)

    with open(input\_file, 'rb') as f:

        file\_data = f.read()

    encrypted\_data = public\_key.encrypt(

        file\_data,

        padding.OAEP(

            mgf=padding.MGF1(algorithm=hashes.SHA256()),

            algorithm=hashes.SHA256(),

            label=None

        )

    )

    with open(output\_file, 'wb') as f:

        f.write(encrypted\_data)

def decrypt\_file(private\_key, input\_file, output\_file):

    private\_key = serialization.load\_pem\_private\_key(private\_key, password=None)

    with open(input\_file, 'rb') as f:

        encrypted\_data = f.read()

    decrypted\_data = private\_key.decrypt(

        encrypted\_data,

        padding.OAEP(

            mgf=padding.MGF1(algorithm=hashes.SHA256()),

            algorithm=hashes.SHA256(),

            label=None

        )

    )

    with open(output\_file, 'wb') as f:

        f.write(decrypted\_data)

def main():

    private\_key = None

    public\_key = None

    while True:

        print("Menu:")

        print("1. Generate Key Pair")

        print("2. Sign a File")

        print("3. Verify Signature")

        print("4. Encrypt File")

        print("5. Decrypt File")

        print("6. Quit")

        choice = input("Enter your choice: ")

        if choice == "1":

            private\_key, public\_key, key\_pair = generate\_key\_pair()

            print("Key pair generated.")

            print("Private Key:\n", key\_pair.private\_bytes(

                encoding=serialization.Encoding.PEM,

                format=serialization.PrivateFormat.TraditionalOpenSSL,

                encryption\_algorithm=serialization.NoEncryption()).decode('utf-8'))

            print("Public Key:\n", public\_key.decode('utf-8'))

        elif choice == "2":

            if private\_key is None:

                print("Please generate a key pair first.")

            else:

                input\_file = input("Enter the name of the file to sign: ")

                if os.path.exists(input\_file):

                    with open(input\_file, 'r') as f:

                        message = f.read()

                    signature = sign\_message(private\_key, message)

                    with open("signature.bin", "wb") as signature\_file:

                        signature\_file.write(signature)

                    print("Signature generated and saved as signature.bin")

                else:

                    print("File not found.")

        elif choice == "3":

            if public\_key is None:

                print("Please generate a key pair first.")

            else:

                input\_file = input("Enter the name of the file to verify: ")

                if os.path.exists(input\_file):

                    with open(input\_file, 'r') as f:

                        message = f.read()

                    with open("signature.bin", "rb") as signature\_file:

                        signature = signature\_file.read()

                    if verify\_signature(public\_key, message, signature):

                        print("Signature is valid.")

                    else:

                        print("Signature is not valid.")

                else:

                    print("File not found.")

        elif choice == "4":

            if public\_key is None:

                print("Please generate a key pair first.")

            else:

                input\_file = input("Enter the name of the file to encrypt: ")

                output\_file = input("Enter the name of the encrypted file: ")

                if os.path.exists(input\_file):

                    encrypt\_file(public\_key, input\_file, output\_file)

                    print("File encrypted and saved.")

                else:

                    print("File not found.")

        elif choice == "5":

            if private\_key is None:

                print("Please generate a key pair first.")

            else:

                input\_file = input("Enter the name of the file to decrypt: ")

                output\_file = input("Enter the name of the decrypted file: ")

                if os.path.exists(input\_file):

                    decrypt\_file(private\_key, input\_file, output\_file)

                    print("File decrypted and saved.")

                else:

                    print("File not found.")

        elif choice == "6":

            break

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**OUTPUT(PROGRAM):**

A screenshot of a computer screen

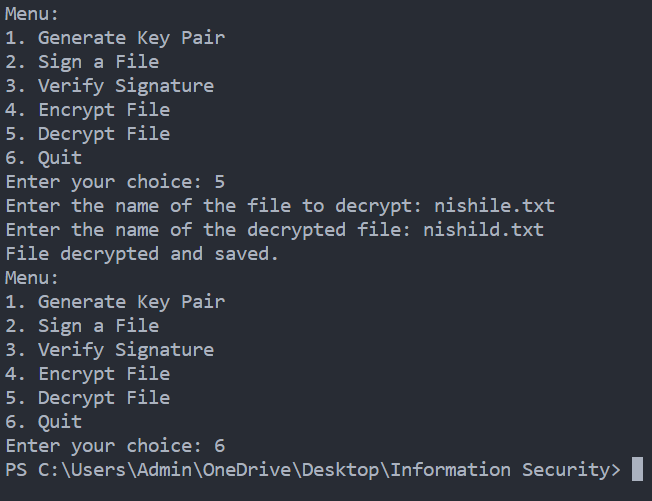
Description automatically generated

A computer screen shot of white letters

Description automatically generated

A screenshot of a computer program

Description automatically generated



**OUTPUT(CRYPTOOL):**

**CRYPTANALYSIS:**

Cryptanalysis for RSA, particularly in the context of digital signature generation and verification, involves examining potential vulnerabilities and attacks against the system. RSA is widely regarded as a robust algorithm, but it's essential to understand the potential weaknesses and how to mitigate them.

Cryptanalysis of RSA for Digital Signature Generation and Verification

**Brute Force Attack**: One common cryptanalysis technique is to perform a brute force attack by systematically trying all possible private key combinations to forge a valid digital signature. This attack becomes increasingly difficult as the key size grows, making longer key lengths more resistant to brute force attacks.

**Padding Oracle Attack**: In digital signature schemes, attackers may exploit padding oracles to gain information about the private key. Padding schemes like PKCS#1 v1.5 are known to be vulnerable to this type of attack, which can reveal the private key if improperly implemented.

**Timing Attacks:** Cryptanalysts may leverage timing information to extract the private key. By measuring the time it takes to perform signature operations, an attacker can gain insights into the private key, especially when dealing with vulnerable software or hardware implementations.

**Adaptive Chosen-Message Attack:** Attackers can use adaptive chosen-message attacks to manipulate the signing oracle. In this scenario, the attacker submits messages for signing and observes the corresponding signatures. Over time, this could lead to the extraction of the private key.

**Random Number Generator (RNG) Vulnerabilities:** Weaknesses in the random number generator used for key pair generation can compromise the entire RSA system. If the generated prime numbers are not sufficiently random, an attacker might be able to predict them, which can lead to private key recovery.

**Small Private Exponent Attack:** If the private exponent is too small, it may be vulnerable to attacks like Wiener's attack. Cryptanalysts can use continued fraction expansions to recover the private key when the private exponent is small relative to the modulus.

**Fault Injection Attacks:** Attackers can manipulate the RSA operations by introducing faults in the computation process. By injecting errors, they may gain insights into the private key. This type of attack often requires physical access to the hardware.

**Quantum Computing Threat:** As quantum computers advance, they pose a potential threat to RSA. Shor's algorithm, when executed on a powerful quantum computer, may factor large RSA modulus efficiently, rendering RSA insecure. This emphasizes the need for post-quantum cryptographic solutions.

**APPLICATIONS:**

**Email Communication:**

**Sender's Authentication Only**: In email communication, RSA digital signatures ensure that emails are genuinely sent by the claimed sender, adding a layer of trust to the message. The recipient can verify the sender's identity based on the digital signature.

**Message Confidentiality and Sender's Authentication:** Combining RSA digital signatures with encryption (such as PGP or S/MIME) ensures both the sender's authenticity and message confidentiality. The recipient can verify the sender and decrypt the message securely.

**Software Distribution:**

**Sender's Authentication Only**: Software developers use RSA digital signatures to sign software packages. Users can verify that the software has not been tampered with and comes from the authentic source.

**Message Confidentiality and Sender's Authentication**: In cases where software also contains sensitive information, RSA digital signatures can be paired with encryption to ensure both the sender's authenticity and the confidentiality of the software's content.

**Legal Documents:**

**Sender's Authentication Only**: Law firms and notaries use RSA digital signatures to sign legal documents, making them legally binding. Recipients can verify the authenticity of documents in a court of law.

**Message Confidentiality and Sender's Authentication:** For confidential legal documents, combining digital signatures with encryption is essential to safeguard the content and the sender's identity.

**Financial Transactions:**

**Sender's Authentication Only:** In online banking and financial transactions, RSA digital signatures authenticate the sender and the integrity of transaction details. This builds trust in online banking services.

**Message Confidentiality and Sender's Authentication:** When financial data is sensitive, combining digital signatures with encryption ensures both the sender's authenticity and the confidentiality of the financial information.

**Secure File Sharing and Collaboration:**

**Sender's Authentication Only:** Secure file sharing platforms use RSA digital signatures to verify that shared files are unaltered and sent by the claimed user. This enhances the security of shared documents in collaborative environments.

**Message Confidentiality and Sender's Authentication:** For private or confidential documents, combining digital signatures with encryption provides comprehensive protection, ensuring the sender's authenticity and message confidentiality.

**REFERENCES:**

Stallings, W. (2017). *Cryptography and Network Security: Principles and Practice.* Pearson. <https://www.pearson.com/en-us/subject-catalog/p/cryptography-and-network-security-principles-and-practice/P200000003477>