**PANDIT DEENDAYAL ENERGY UNIVERSITY**

**SCHOOL OF TECHNOLOGY**



**Course: Information Security Lab**

**Course Code: 20CP304P**

**LAB MANUAL**

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

**Semester 5**

|  |  |
| --- | --- |
| **Submitted To:** | **Submitted By:** |
| Dr. Hargeet Kaur | Ravi Jamanbhai Makwana |
|  | 21BCP418 |
|  | G12 batch |

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **List of experiments** | **Date** | **Sign** |
| **1** | Download and practice Cryptool. | 7/8/23 |  |
| **2** | Study and Implement program for Caeser Cipher with Encryption, Decryption functions. | 7/8/23 |  |
| **3** | Study and implement a program for Transposition (Columnar) Cipher to encrypt and decrypt the message. | 14/8/23 |  |
| **4** | Study and implement a program for Rail Fence Transposition (Columnar) Cipher to encrypt and decrypt a message | 21/8/23 |  |
| **5** | Study and implement a program for Vigenère Cipher to encrypt and decrypt a message | 28/8/23 |  |
| **6** | Study and implement a program for Playfair Cipher to encrypt and decrypt a message | 4/9/23 |  |
| **7** | Study and implement a program for Hill Cipher to encrypt and decrypt a message | 16/10/23 |  |
| **8** | Use Crypto++ library to implement encryption and decryption of different block ciphers | 30/10/23 |  |
| **9** | Study and implement RSA encryption and decryption functions. | 30/10/23 |  |
| **10** | Use RSA for generation and verification of digital signature on file. | 30/10/23 |  |

**Experiment-1**

**Aim:** Download and Practice Cryptool.

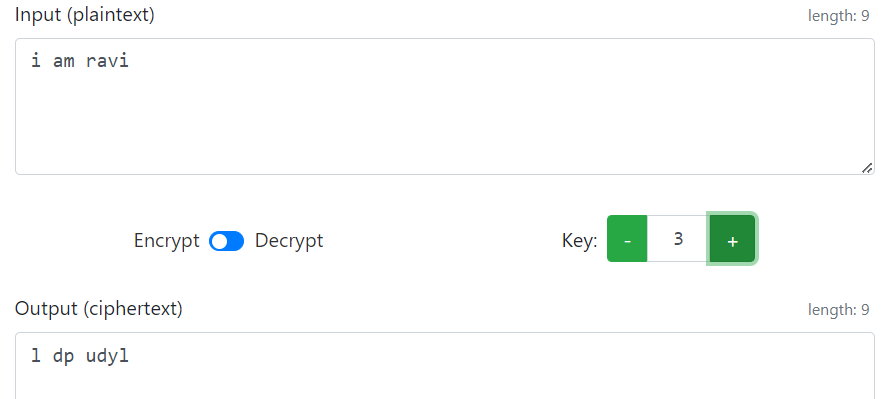
**Introduction:**

Cryptool is an open-source project that is a free e-learning software for illustrating cryptographic and cryptanalytic concepts.

Cryptool stands as a dynamic and adaptable software suite, positioned at the forefront of modern cryptography investigation and analysis. Its intuitive interface and meticulously crafted cryptographic tools empower both newcomers and seasoned professionals to immerse themselves in the captivating realm of encryption, decryption, and cryptographic analysis. Whether one's goal is to master the foundational concepts of cryptography, evaluate algorithm security, or untangle intricate historical ciphers, Cryptool serves as an essential platform for hands-on exploration, educational pursuits, and scholarly research. By seamlessly melding theoretical understanding with practical engagement, Cryptool emerges as a vital companion for unravelling the intricacies of cryptographic techniques, reinforcing digital security, and cultivating a deeper comprehension of the intricate craft of code crafting and deciphering.

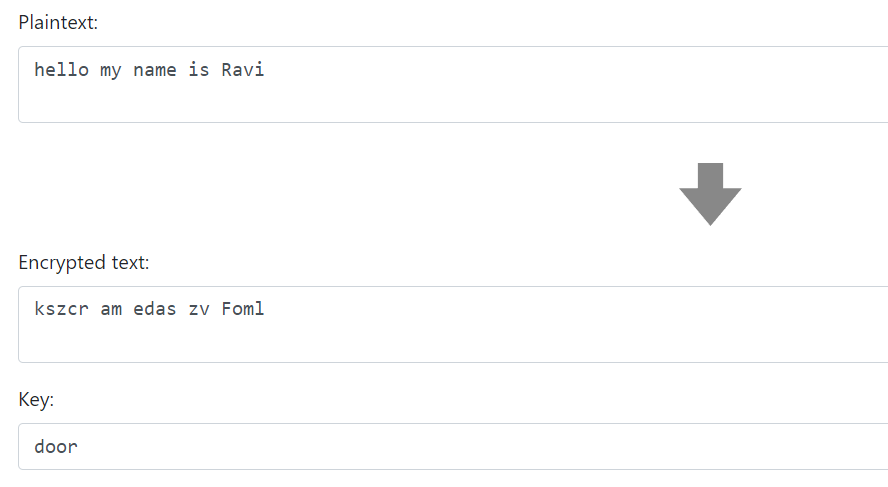
**PRACTICE**

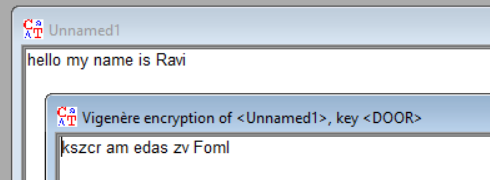
1. **Ceasar Cipher:**



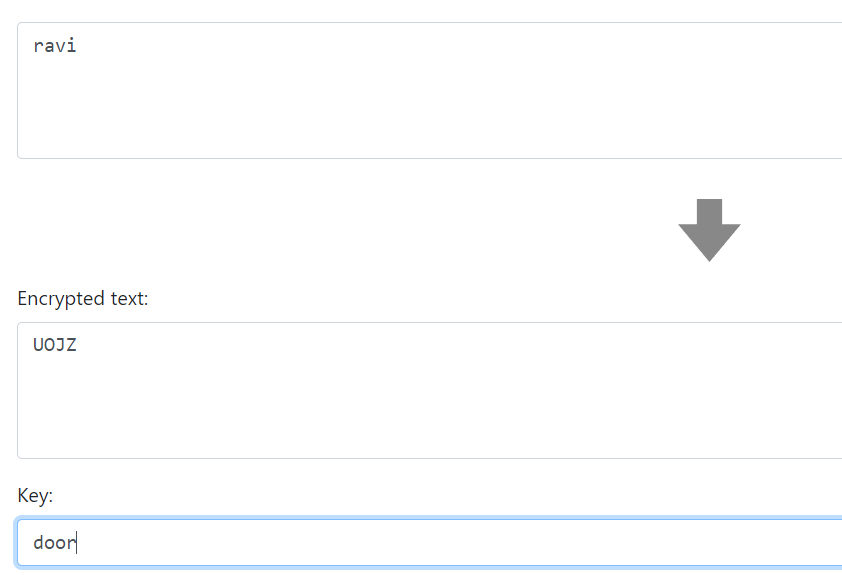


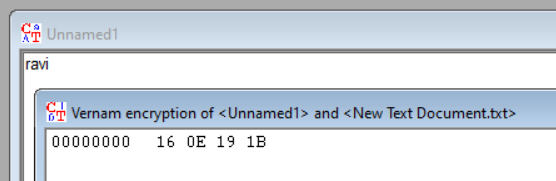
**2) Vigenère Cipher:**



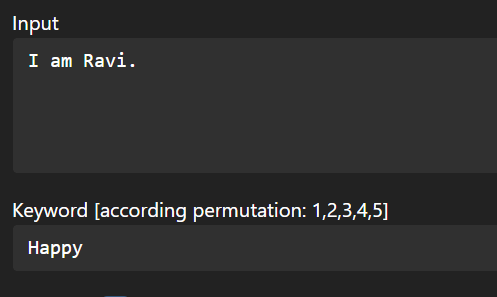


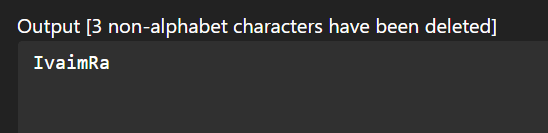
**3) Vernam Cipher:**

****

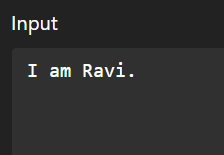
****

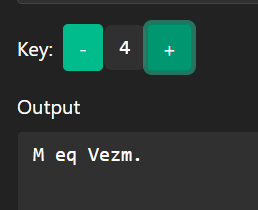
**4) Simple Columnar Transposition Cipher:**





**5) Mono alphabetic Substitution Cipher:**





**Experiment-2**

**Aim:** Study and Implement program for Caeser Cipher with Encryption, Decryption functions.

**Introduction:**

In cryptography, a Caesar cipher, also known as Caesar's cipher, the shift cipher, Caesar's code, or Caesar shift, is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. For example, with a left shift of 3, D would be replaced by A, E would become B, and so on. The method is named after Julius Caesar, who used it in his private correspondence.

It works as:

* **Key (Shift Value):** A key is chosen, which determines the number of positions each letter is shifted. For example, if the key is 3, then each letter in the plaintext will be replaced with the letter that is 3 positions down the alphabet.
* **Encryption:** To encrypt a message, each letter in the plaintext is replaced by the letter located a certain number of positions down the alphabet. If the shift value is positive (as in the original Caesar Cipher), the letters wrap around from the end of the alphabet to the beginning if necessary.

For example, using a key of 3:

'A' becomes 'D’, ‘B' becomes 'E', 'C' becomes 'F' ….'Z' becomes 'C'

* **Decryption:** To decrypt a message encrypted with the Caesar Cipher, you perform the reverse process. Each letter in the ciphertext is shifted back by the same key value to reveal the original plaintext.

**Program (Source Code):**

#include <bits/stdc++.h>

using namespace std;

string cipher(string P, int key){

    string C;

    // converting original text to lowercase

    for (int i=0;i<P.length();i++){

        P[i] = tolower(P[i]);

    }

    // converting to cipher text

    for (int i=0;i<P.length();i++){

        char ch;

        if (P[i] != ' '){

            ch = (int(P[i]) + (key % 26) - 97) % 26 + 97;

        }

        else{

            ch = ' ';

        }

        C += ch;

    }

    return C;

}

string decipher(string P, int key){

    string C;

    // converting original text to lowercase

    for (int i=0;i<P.length();i++){

        P[i] = tolower(P[i]);

    }

    // converting to cipher text

    for (int i=0;i<P.length();i++){

        char ch;

        if (P[i] != ' '){

            ch = (int(P[i]) - (key % 26) - 97 + 26) % 26 + 97;

        }

        else{

            ch = ' ';

        }

        C += ch;

    }

    return C;

}

int main(){

    string plainText = "my name is ravi";

    int key = 3;

    string cipherText = cipher(plainText, key);

    cout<<"Encrypted Text: "<<cipherText;

    cout<<"\n";

    string decyp = decipher(cipherText, key);

    cout<<"Decrypted Text: "<<decyp;

    return 0;

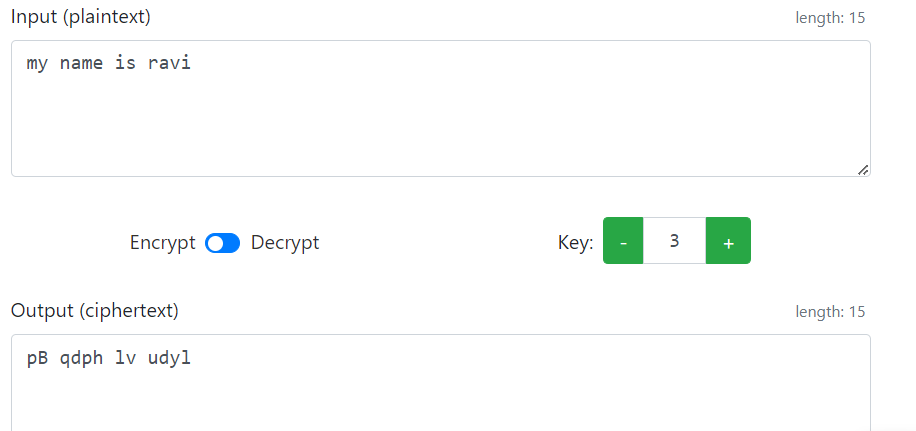
}

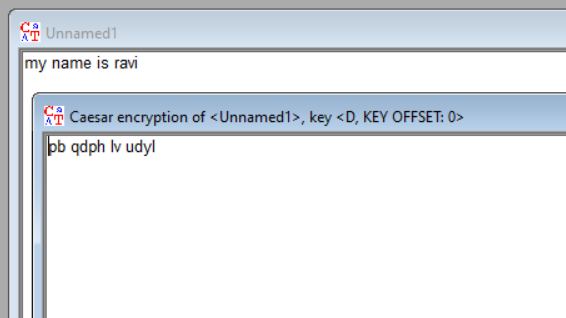
**Output (Program):**



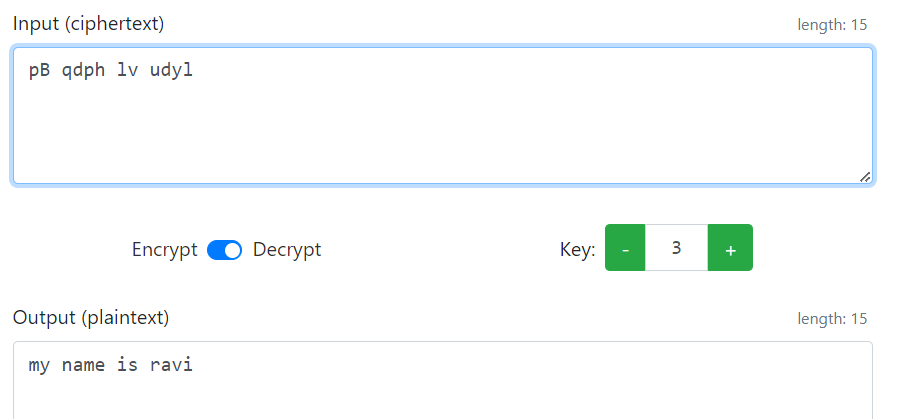
**Output (Cryptool):**

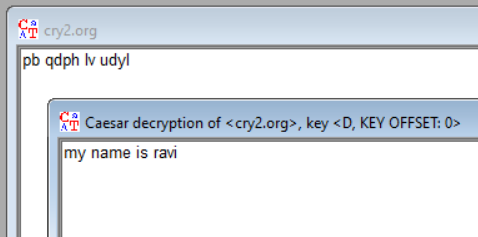
**Encryption:**





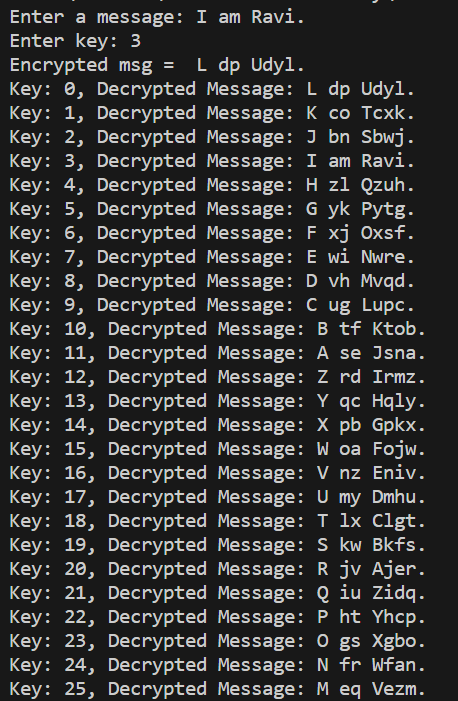
**Decryption:**

****



**Cryptanalysis:**

**Brute force Analysis:**



**2. Frequency Analysis:** Even though the Caesar cipher obscures the original letters, it does not change the frequency distribution of letters in the text. For example, in English, the letter "E" is the most used letter. An attacker can analyze the frequency of letters in the ciphertext and make educated guesses about the key based on the known frequency distribution of letters in the language.

**3. Known-Plaintext Attack:** If the attacker has some knowledge of the plaintext content or can make educated guesses about parts of the original text, they can use this information to narrow down possible keys. By aligning the known plaintext with the ciphertext, they can deduce the key.

**Applications:**

The Caesar cipher, although not suitable for strong security, can still have applications in:

* Education: It is a great tool to introduce beginners to the concepts of encryption and decryption in cryptography.
* Puzzles and Challenges: In recreational settings, it can add an element of mystery to puzzles, games, or coding challenges.
* Historical Context: It can be used in historical re-enactments to demonstrate ancient cryptographic techniques.
* Introductory Coding: It can be used to teach basic programming skills involving strings and loops.
* Steganography: As a component in larger steganographic methods where information is hidden within other media.

**References:**

1. W3Schools
2. GeeksforGeeks

**Experiment-3**

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

**Introduction:**

Columnar transposition is a cryptographic technique used to encrypt plaintext by rearranging the letters or characters in a specific way based on a keyword or key phrase. It falls under the category of classical or historical encryption methods and relies on a simple permutation operation.

Here is how the columnar transposition encryption process works:

1. Key Selection: Choose a keyword or key phrase that will be used for encryption. This keyword determines the order in which columns are written during the transposition process.

2. Creating the Transposition Grid: Write the keyword horizontally at the top of a grid. Each letter in the keyword becomes the label for a column. Arrange the letters of the keyword in alphabetical order without repeating any letters. Fill in the remaining cells of the grid with the rest of the alphabet letters in order.

3. Encryption: Write the plaintext message row by row into the grid, filling the cells in order. If a row is not filled, pad it with dummy characters.

4. Reading the Cipher: Read the ciphertext column by column from the transposition grid, starting with the columns corresponding to the keyword letters in alphabetical order. This creates the encrypted message.

Since the arrangement of the columns is determined by the keyword, changing the keyword will result in a different permutation of the columns, and thus a different ciphertext.

**Program (Source Code):**

#include <bits/stdc++.h>

using namespace std;

string encrypt(string plainText, string keyword){

    string EncryptedText;

    double columns = keyword.length();

    double rows = ceil(plainText.length() / columns);

    char matrix[int(rows)][int(columns)];

    int stringIndexPointer = 0;

    //inserting characters into the table(row-wise)

    for (int i=0;i<rows;i++){

        for (int j=0;j<columns;j++){

            if (stringIndexPointer < plainText.length()){

                matrix[i][j] = plainText[stringIndexPointer];

                stringIndexPointer++;

            }

            //when plaingText traversal is over but table is still left to fill

            //fill it with '\_'

            else{

                matrix[i][j] = '\_';

            }

        }

    }

    // //for printing the table

    // for (int i=0;i<rows;i++){

    //     for (int j=0;j<columns;j++){

    //         cout<<matrix[i][j]<<" ";

    //     }

    //     cout<<"\n";

    // }

    //finding the index of highest ASCII value character in the keyword (used later)

    int highestASCIIValueindex = 0;

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

        if (keyword[i] >= keyword[highestASCIIValueindex]){

            highestASCIIValueindex = i;

        }

    }

    //for deciding order of reading the columns//

    //order will be stored in the below array

    int orderArr[keyword.length()];

    //for keeping track whether character is traversed or not

    int hashArr[keyword.length()] = {0};

    //for inserting into the orderArr

    int pointerForInsert = 0;

    //finding smallest ascii value in the keyword and storing its index in the orderArr each time.

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

        int smallestASCIIValueIndex = highestASCIIValueindex;

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

            if (keyword[j] <= keyword[smallestASCIIValueIndex] && hashArr[j] == 0){

                smallestASCIIValueIndex = j;

            }

        }

        orderArr[pointerForInsert++] = smallestASCIIValueIndex;

        hashArr[smallestASCIIValueIndex] = 1;

    }

    //forming the decryptedText

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

        int columnNumber = orderArr[i];

        for (int j=0;j<rows;j++){

            if(matrix[j][columnNumber] != '\_'){

                EncryptedText += matrix[j][columnNumber];

            }

        }

    }

    return EncryptedText;

}

string decrypt(string encryptedText, string keyword){

    string decryptedText;

    double columns = keyword.length();

    double rows = ceil(encryptedText.length() / columns);

    char matrix[int(rows)][int(columns)];

    //inserting characters into the table (column-wise)

    int stringIndexPointer = 0;

    for (int i=0;i<columns;i++){

        for (int j=0;j<rows;j++){

            if (stringIndexPointer < encryptedText.length()){

                matrix[j][i] = encryptedText[stringIndexPointer];

                stringIndexPointer++;

            }

            //when plaingText traversal is over but table is still left to fill

            //fill it with '\_'

            else{

                matrix[j][i] = '\_';

            }

        }

    }

    // //for printing the table

    // for (int i=0;i<rows;i++){

    //     for (int j=0;j<columns;j++){

    //         cout<<matrix[i][j]<<" ";

    //     }

    //     cout<<"\n";

    // }

    //finding the index of highest ASCII value character in the keyword (used later)

    int highestASCIIValueindex = 0;

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

        if (keyword[i] >= keyword[highestASCIIValueindex]){

            highestASCIIValueindex = i;

        }

    }

    //for deciding order of reading the columns//

    //order will be stored in the below array

    int orderArr[keyword.length()];

    //for keeping track whether character is traversed or not

    int hashArr[keyword.length()] = {0};

    //for inserting into the orderArr

    int pointerForInsert = 0;

    //finding smallest ascii value in the keyword and storing its index in the orderArr each time.

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

        int smallestASCIIValueIndex = highestASCIIValueindex;

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

            if (keyword[j] <= keyword[smallestASCIIValueIndex] && hashArr[j] == 0){

                smallestASCIIValueIndex = j;

            }

        }

        orderArr[pointerForInsert++] = smallestASCIIValueIndex;

        hashArr[smallestASCIIValueIndex] = 1;

    }

    //but since reading order (for decryption) will be different, actual order will be different

    //finding actual order

    int actualOrder[keyword.length()];

    int pointer = 0;

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

        //finding i in orderArr, and appending its index to actualOrder

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

            if (i == orderArr[j]){

                // j will give the actual order

                actualOrder[pointer++] = j;

            }

        }

    }

    //forming the decryptedText

    for (int i=0;i<rows;i++){

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

            int columnNumber = actualOrder[j];

            if (matrix[i][columnNumber] != '\_'){

                decryptedText += matrix[i][columnNumber];

            }

        }

    }

    return decryptedText;

}

int main(){

string plainText = "My name is Ravi";

string keyword = "PDEU";

    cout<<"Plain Text: "<<plainText;

    cout<<"\nKeyword: "<<keyword;

    string encryptedText = encrypt(plainText, keyword);

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

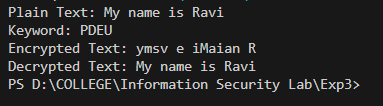
    string decryptedText = decrypt(encryptedText, keyword);

    cout<<"\nDecrypted Text: "<<decryptedText;

    return 0;

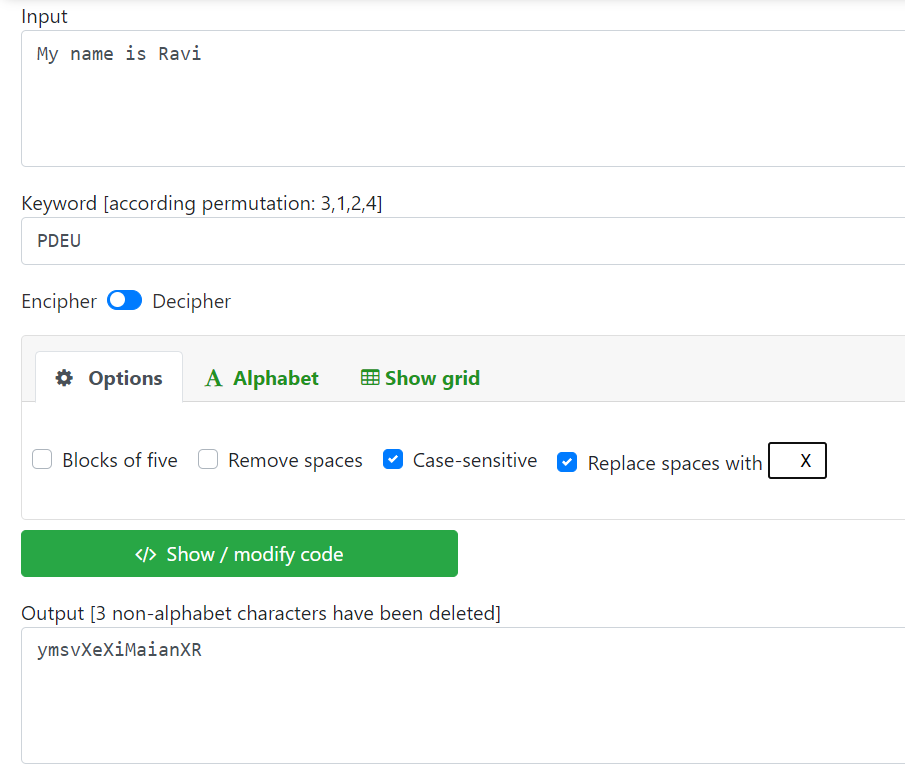
}

**Output (Program):**

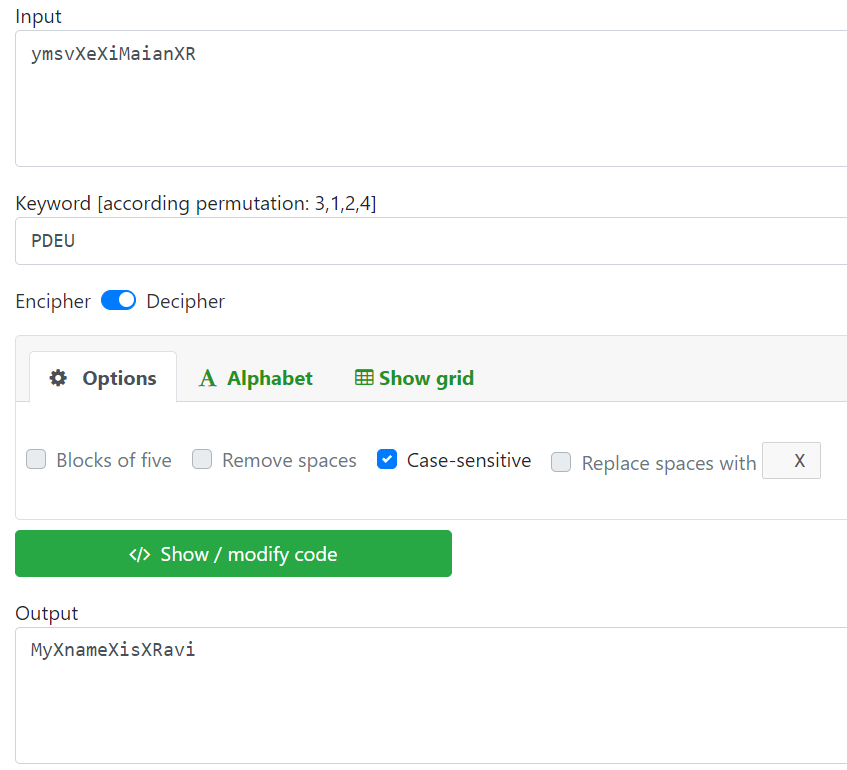
****

**Output (Cryptool):**

**Encryption:**

****

**Decryption:**



**Cryptanalysis:**

**1. Brute Force Attack:** This can be done if we know the length of the keyword. This involves trying all possible column orders based on the key length. Since the key is a permutation of column numbers, the number of possible keys can grow rapidly with the number of columns. However, for small key lengths, this attack can be feasible.

**2. Dictionary Attack:** The columnar transposition cipher is almost always keyed with a word or short phrase, so we may not need to test all possible transposition keys, we may only need to test common words. This involves having a large list of dictionary words including place names, famous people, mythological names, historical names etc. From this we generate a text file of possible keys. This method can work if the keyword was one of the words that we included in the dictionary, but it fails to work if the keyword is the mixture of more than one word.

**3. Hill Climbing:** We first assume the key length is 10, then choose a random starting keyword of this length. This is called the parent key. Child keys are generated by making random swaps in the parent keyword, and if any of the swaps lead to an increased fitness, we replace the parent with the child that beat it. If after many tries the correct key is not found, it is time to increment the key length to 11 and rerun everything.

**Applications:**

Columnar transposition, despite its lack of robust security in modern contexts, can still find applications in certain scenarios or as part of more complex encryption methods. Here are a few potential applications:

1. **Educational Purposes**: Columnar transposition is often used to teach the basics of encryption and decryption. It helps students understand fundamental concepts like permutation and the importance of encryption keys.
2. **Puzzle and Games**: Columnar transposition can be used as a basis for creating puzzles or games that involve solving simple cryptographic challenges. These can be entertaining and educational for people interested in cryptography or logic puzzles.
3. **Historical Reenactments**: Since columnar transposition is a classical encryption method, it can be used in historical reenactments, escape room activities, or events that aim to showcase historical methods of encryption and codebreaking.
4. **Obfuscation and Basic Protection**: While not suitable for serious security applications, columnar transposition can still be used for basic obfuscation of text. For example, it can make casual reading of a message more challenging for unintended recipients who are not familiar with the method.
5. **Part of Hybrid Encryption**: In some cases, columnar transposition could be used as one step in a multi-layered encryption process. By combining it with other encryption techniques, it might contribute to increasing the overall complexity of encryption.
6. **Code Words and Simple Encodings**: In non-critical applications, columnar transposition could be used to create code words or encodings for fun or artistic purposes. It can add a layer of mystique to messages without requiring highly secure encryption.
7. **Text Transformations**: Columnar transposition can be used as a text transformation method in data processing or manipulation. For example, it might be used to shuffle the order of words in a sentence for creative writing purposes.

It is worth noting that for securing sensitive information, modern encryption methods like AES (Advanced Encryption Standard) are highly recommended due to their well-established security properties and resistance to various attacks. Columnar transposition is considered relatively weak against sophisticated cryptanalysis techniques, so caution is necessary when using it for any security-related tasks.

**References:**

* 1. W3Schools

1. GeeksforGeeks
2. www.practicalcryptography.com

**Experiment-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 have seen in encryption, 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 have 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 (Source Code):**

//Note: in railfence cipher, spaces ARE considered, so do not truncate them.

#include <bits/stdc++.h>

using namespace std;

//logic: make a matrix of rail \* length(plainText) dimensions, and traverse the string and fill the matrix

string encrypt(string plainText, int rail){

    string encryptedText;

    int len = plainText.length();

    char matrix[rail][len];

    //filling the matrix with '-'

    for (int i=0;i<rail;i++){

        for (int j=0;j<len;j++){

            matrix[i][j] = '-';

        }

    }

    int flag = 1;   //flag = 1 means increment the value of  rows, flag = 0 means decrement the value of rows. [Initially set to 1]

    int row = 0;

    for (int i=0;i<len;i++){    // traversing the plainText. [value of i will be the column number]

        //setting the flag (to choose whether to increment or decrement)

        if (row == 0){

            flag = 1;

        }

        else if (row == (rail-1)){

            flag = 0;

        }

        else{

            //do nothing

        }

        //push the value

        matrix[row][i] = plainText[i];

        //choosing the next row value in the matrix [update value of row]

        if (flag == 1){

            row++;

        }

        else{

            row--;

        }

    }

    // //printing the matrix

    // cout<<"\nMatrix:\n";

    // for (int i=0;i<rail;i++){

    //     for (int j=0;j<len;j++){

    //         cout<<matrix[i][j]<<" ";

    //     }

    //     cout<<"\n";

    // }

    //making the encrypted text

    for (int i=0;i<rail;i++){

        for (int j=0;j<len;j++){

            if (matrix[i][j] != '-'){

                encryptedText += matrix[i][j];

            }

        }

    }

    return encryptedText;

}

//logic: make the matrix and fill the places (where the encrypted text characters will be placed) with \* and then traverse the matrix again and fill it with ciphertext wherever there are \*

string decrypt(string encryptedText, int rail){

    string decryptedText;

    int len = encryptedText.length();

    char matrix[rail][len];

    //filling the matrix with '-'

    for (int i=0;i<rail;i++){

        for (int j=0;j<len;j++){

            matrix[i][j] = '-';

        }

    }

    //filling the places in the matrix with '\*' wherever the encrypted text characters need to be put (same way as done in encryption)

    int flag = 1;

    int row = 0;

    for (int i=0;i<len;i++){

        if (row == 0){

            flag = 1;

        }

        else if (row == rail-1){

            flag = 0;

        }

        else{

            //do nothing

        }

        matrix[row][i] = '\*';

        if (flag == 1){

            row++;

        }

        else{

            row--;

        }

    }

    // //printing the matrix

    // cout<<"\nMatrix:\n";

    // for (int i=0;i<rail;i++){

    //     for (int j=0;j<len;j++){

    //         cout<<matrix[i][j]<<" ";

    //     }

    //     cout<<"\n";

    // }

    //traversing through the matrix and filling it with the encrypted text wherever there is a '\*'

    int stringPointer = 0;

    for (int i=0;i<rail;i++){

        for (int j=0;j<len;j++){

            if (matrix[i][j] == '\*'){

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

            }

        }

    }

    //forming the decrypted text the same way we put the '\*' (traversing method is same) [this is the same way encryption was done, but now we're reading it instead of writing it]

    flag = 1;

    row = 0;

    for (int i=0;i<len;i++){

        if (row == 0){

            flag = 1;

        }

        else if (row == rail-1){

            flag = 0;

        }

        else{

            //do nothing

        }

        decryptedText += matrix[row][i];

        if (flag == 1){

            row++;

        }

        else{

            row--;

        }

    }

    return decryptedText;

}

int main(){

    string plainText = "My name is Ravi and I study in PDEU";

    int rail = 4;

    cout<<"Plain Text: "<<plainText;

    cout<<"\nRail: "<<rail;

    string encryptedText = encrypt(plainText, rail);

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

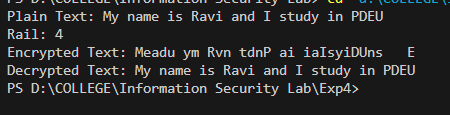
    string decryptedText = decrypt(encryptedText, rail);

    cout<<"\nDecrypted Text: "<<decryptedText;

    return 0;

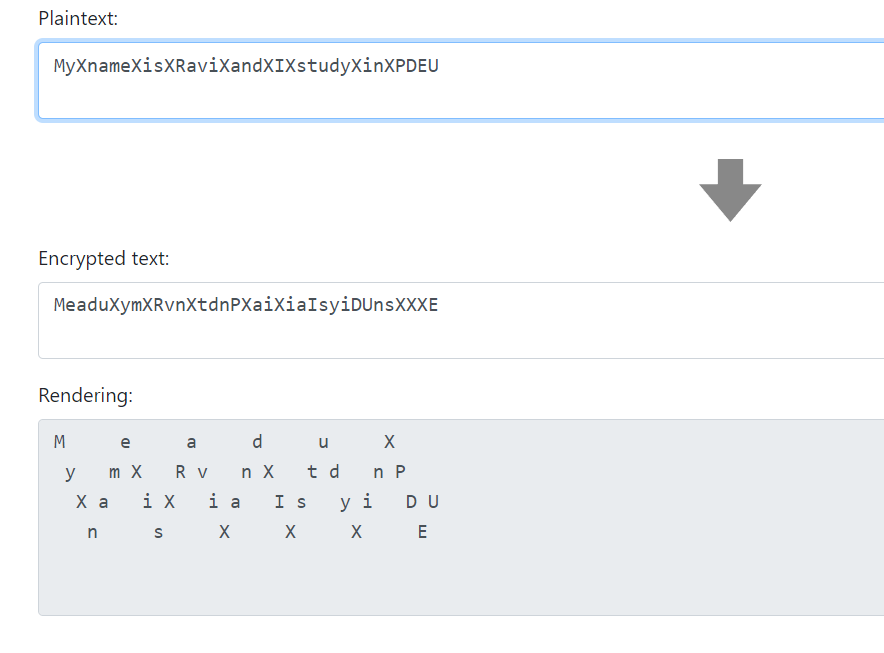
}

**Output (Program):**

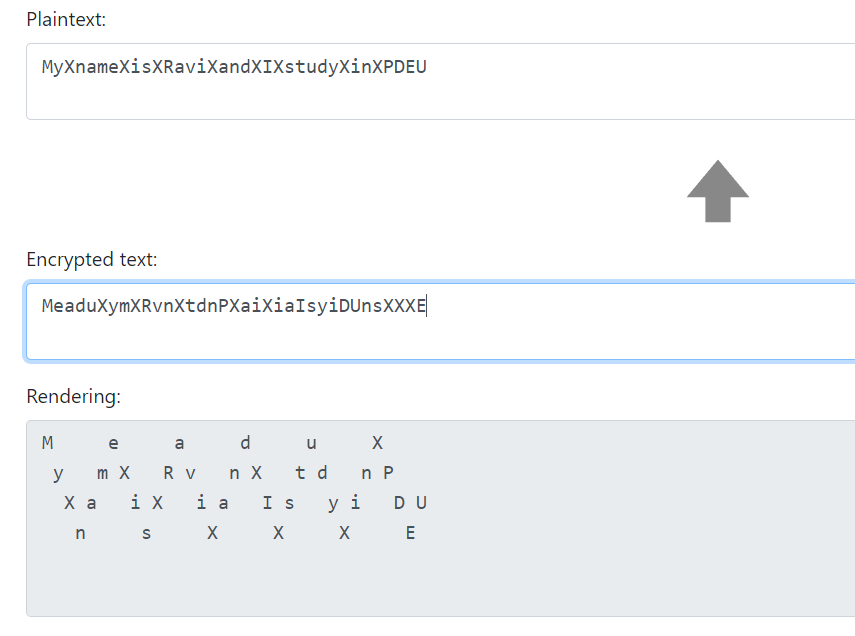
****

**Output (Cryptool):**

**Encryption:**

****

**Decryption:**



**Cryptanalysis:**

1. **Brute Force Attack:** The cipher's key is N, the number of rails. If N is known, the ciphertext can be decrypted by using the above algorithm. Values of N equal to or greater than L, the length of the ciphertext, are not usable, since then the ciphertext is the same as the plaintext. Therefore, the number of usable keys is low, allowing the brute-force attack of trying all possible keys. As a result, the rail-fence cipher is considered weak. We need a way of figuring out which of the keys results in the most English like plaintext after decryption. The key that results in a decryption with the highest likelihood of being English text is most probably the correct key. So, the method used is to take the ciphertext, try decrypting it with each key, then see which decryption looks the best.

**Applications:**

The Rail Fence cipher, despite its simplicity and susceptibility to cryptanalysis, can still find some limited applications where strong security is not a primary concern. Here are a few examples of where the Rail Fence cipher might be used:

1. **Educational Purposes:** The Rail Fence cipher is often used as an introductory example in cryptography courses. It helps students understand basic concepts of encryption, transposition ciphers, and the importance of key management.
2. **Puzzle and Games:** The Rail Fence cipher can be employed as part of puzzles, games, or challenges in recreational settings. It can add an element of fun and intrigue to activities where participants need to decipher messages.
3. **Steganography:** While not secure for cryptographic purposes, the Rail Fence cipher could be used as a form of steganography, where messages are hidden in plain sight within seemingly innocent communication. This could be a casual way of conveying messages without attracting attention.
4. **Historical Interest:** The Rail Fence cipher has a historical significance as one of the early encryption techniques. It might be used in reenactments or historical educational settings to demonstrate how encryption was practiced in the past.
5. **Artistic Expression:** Some artists and writers might use the Rail Fence cipher to encode messages within their work, adding an extra layer of meaning or playfulness to their creations.
6. **Basic Communication:** In scenarios where very, basic security is required (e.g., leaving a note for a family member or friend), the Rail Fence cipher could be used to prevent casual readers from understanding the message immediately.
7. **Ciphers for Children:** The Rail Fence cipher's simple structure makes it suitable for introducing young children to the concept of encryption in a playful manner.

**References:**

* + 1. GeeksforGeeks
    2. www.practicalcryptography.com
    3. crypto.interactive-maths.com
    4. en.wikipedia.org

**Experiment-5**

**Aim:** Study and Implement a program for Vigenère Cipher to encrypt and decrypt the message.

**Introduction:**

Vigenère 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.

* 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 Caesar 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. For generating key, the given keyword is repeated in a circular manner until it matches the length of the plain text.

**Encryption:**

The first letter of the plaintext is combined with the first letter of the key. Then, the intersection of column of plain text’s character and row of key’s character is seen in the Vigenère table. That is the ciphertext’s character. Similarly, the second letter of the plaintext is combined with the second letter of the key. This process continues continuously until the plaintext is finished.

**Decryption:**

Decryption is done by the row of keys in the Vigenère table. First, select the row of the key letter, find the ciphertext letter's position in that row, and then select the column label of the corresponding ciphertext as the plaintext.

**Another method** of encryption and decryption is when the Vigenère table is not given. The encryption and decryption are done by Vigenère algebraically formula in this method (convert the letters (A-Z) into the numbers (0-25)).

**Formula of encryption is,**

Ei = (Pi + Ki) mod 26

**Formula of decryption is,**

Di = (Ei - Ki) mod 26

If any case (Di) value becomes negative (-ve), in this case, we will add 26 in the negative value.

**Where:**

E denotes the encryption.

D denotes the decryption.

P denotes the plaintext.

K denotes the key.

**Program (Source Code):**

//In this program, spaces are truncated and characters are converted to lowercase

#include <bits/stdc++.h>

using namespace std;

string encrypt(string plainText, string key){

    string encryptedText;

    string keyword;

    int len = plainText.length();

    //make keyword from key (make it the same length as the plainText)

    int repeat = ceil(double(len)/double(key.length()));

    for (int i=0;i<repeat;i++){

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

            if (keyword.length() < len){

                keyword += key[j];

            }

        }

    }

    //making encryptedText (by adding key and plainText)

    for (int i=0;i<len;i++){

        int ch = (((plainText[i] - 'a') + (keyword[i] - 'a'))%26) + 'a';

        encryptedText += char(ch);

    }

    return encryptedText;

}

string decrypt(string encryptedText, string key){

    string decryptedText;

    string keyword;

    int len = encryptedText.length();

    //make keyword from key (make it the same length as the encryptedText)

    int repeat = ceil(double(len)/double(key.length()));

    for (int i=0;i<repeat;i++){

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

            if (keyword.length() < len){

                keyword += key[j];

            }

        }

    }

    //making decryptedText (by subtracting key from encryptedText)

    for (int i=0;i<len;i++){

        int ch = (((encryptedText[i] - 'a') - (keyword[i] - 'a') + 26)%26) + 'a';

        decryptedText += char(ch);

    }

    return decryptedText;

}

int main(){

    string plainText0 = "mynameisravi";

    string key0 = "pdeu";

    string plainText;

    string key;

    //truncate spaces from key and plainText, and convert to lowercase

    for (int i=0;i<plainText0.length();i++){

        if (plainText0[i] != ' '){

            plainText += tolower(plainText0[i]);

        }

    }

    for (int i=0;i<key0.length();i++){

        if (key0[i] != ' '){

            key += tolower(key0[i]);

        }

    }

    cout<<"Plain Text: "<<plainText;

    cout<<"\nKey: "<<key;

    string encryptedText = encrypt(plainText, key);

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

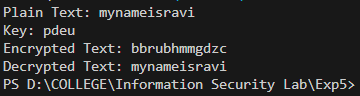
    string decryptedText = decrypt(encryptedText, key);

    cout<<"\nDecrypted Text: "<<decryptedText;

    return 0;

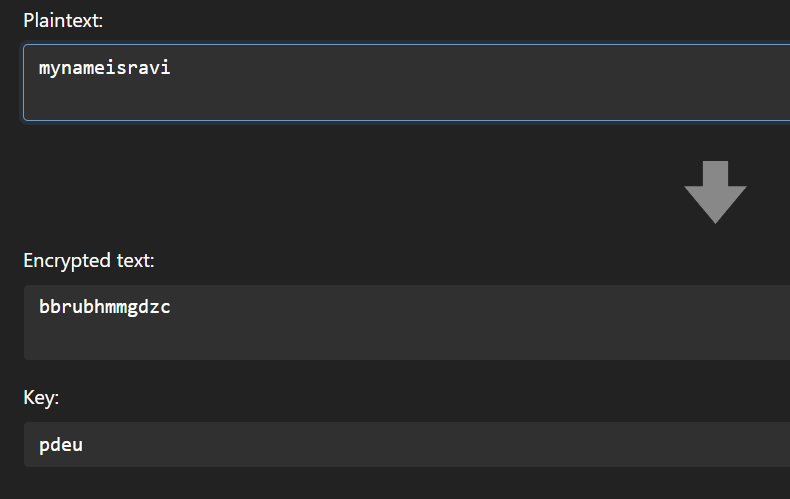
}

**Output (Program):**

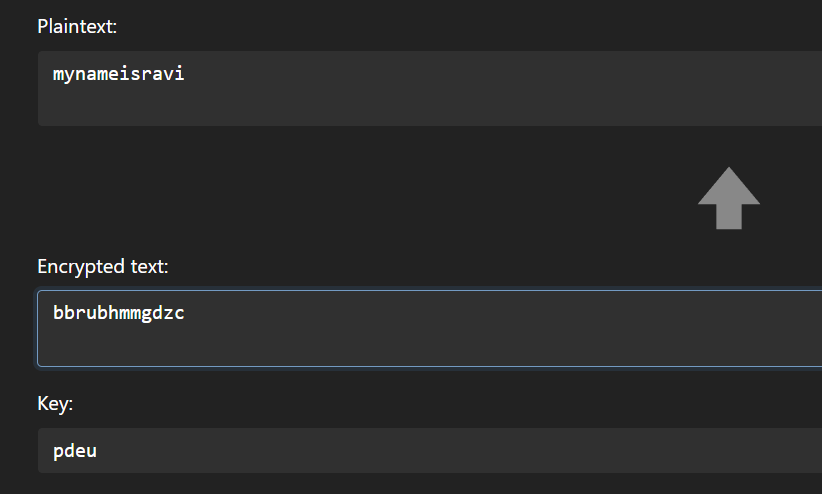
****

**Output (Cryptool):**

**Encryption:**

****

**Decryption:**



**Cryptanalysis:**

There are two main steps involved in breaking a Vigenère cipher. Those two are:

1. **Determine the key length:**

There are 2 ways of finding the key’s length:

**Index of Coincidence:** We start by dividing the message into different groups, each with the same number of characters. Then, we calculate the Index of Coincidence for each group. If the length of your groups matches the length of the keyword, the Index of Coincidence will be higher because the letters in those groups are likely shifted in the same way. In essence, the Index of Coincidence helps us find the right grouping of letters in the encrypted message, which can reveal the length of the keyword and make it easier to decrypt the message.

* **Kasiski Test:** While applying the technique of finding the index of coincidence, you also would have found certain patterns. Using these patterns for finding the length of the key for the message is called the Kasiski test. When we analyse the ciphertext, we get some strong patterns in repetition. This test is based on these observations. We see that two identical segments of the plaintext will be encrypted to the same ciphertext whenever their occurrence in the plaintext is δ positions apart, where δ ≡ 0 (mod m). Conversely, when we observe two identical string segments of the ciphertext, each of length k. There is a good chance that the length of the key may also be k. Each index of the key will act as the key for the corresponding character in the plaintext string.
* **Friedman Test:** It does this by looking at the repeating patterns in the encrypted message. You divide the message into sections of a certain length and then calculate the Index of Coincidence (IC) for each section. If the IC for a particular section matches what you would expect for the language of the message (like English), it suggests that the keyword might be that length. Essentially, it helps you detect when the same part of the keyword is applied to different parts of the message.

1. **Finding the key:**

**Key Elimination:** If the key length is known (or guessed), subtracting the cipher text from itself, offset by the key length, will produce the plain text subtracted from itself, also offset by the key length. If any "probable word" in the plain text is known or can be guessed, its self-subtraction can be recognized, which allows recovery of the key by subtracting the known plaintext from the cipher text.

**Applications:**

The Vigenère cipher is a classical method of encrypting alphabetic text by using a simple form of polyalphabetic substitution. It was originally developed in the 16th century and has been used for various applications throughout history, though it is not considered a secure encryption method today due to its vulnerability to modern cryptanalysis techniques. Here are some historical and modern applications of the Vigenère cipher:

1. Military Communication (Historical): The Vigenère cipher was historically used for secure military communication. It provided a way to encode messages so that only those with the correct key could decipher them.

2. Diplomatic Correspondence (Historical): Diplomats and government officials also used the Vigenère cipher to encode sensitive diplomatic correspondence. It offered a level of security against unauthorized interception of messages.

3. Education: The Vigenère cipher was used as a teaching tool to introduce students to cryptography. Its relatively simple encryption and decryption process made it suitable for educational purposes.

4. Puzzles and Recreational Cryptography: Cryptographers and enthusiasts created puzzles and challenges using the Vigenère cipher as a form of recreational cryptography. This encouraged people to learn about cryptanalysis and decipher encoded messages.

5. Steganography: While not a primary use, the Vigenère cipher can be incorporated into steganography techniques, where the goal is to hide a message within another medium (e.g., an image or audio file) to avoid detection.

It is important to note that the Vigenère cipher is not considered secure for modern cryptographic purposes, as it can be easily broken with modern cryptanalysis techniques.

**References:**

* + - 1. GeeksforGeeks
      2. www.codingninjas.com
      3. en.wikipedia.org

**Experiment-6**

**Aim:** Study and Implement a program for 5\*5 Playfair Cipher.

**Introduction:**

The Playfair cipher or Playfair square or Wheatstone–Playfair cipher is a manual symmetric encryption technique and was the first literal digraph substitution cipher.

The technique encrypts pairs of letters (digraph) instead of single letters as in the simple substitution cipher and rather more complex Vigenère cipher systems then in use. The Playfair is thus significantly harder to break since the frequency analysis used for simple substitution ciphers does not work with it.

**Method of Encryption and Decryption:**

**1) Generate playfair matrix (5X5):** The key square is a 5×5 grid of alphabets that acts as the key for encrypting the plaintext. Each of the 25 alphabets must be unique and one letter of the alphabet (usually J) is omitted from the table (as the table can hold only 25 alphabets). If the plaintext contains J, then it is replaced by I. The initial alphabets in the key square are the unique alphabets of the key in the order in which they appear followed by the remaining letters of the alphabet in order.

**2) Generate Digraph:** The plaintext is split into pairs of two letters (digraphs). If there is an odd number of letters, a X 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.

**3)** **Encryption:**

Take two letters at a time from digraph.

Case 1: Both are in same row: Take the letter to the right of each one (going back to the leftmost if at the rightmost position).

Case 2: Both are in same column: Take the letter below each one (going back to the top if at the bottom).

Case 3: Different row and column: Form a rectangle with the two letters and take the letters on the horizontal opposite corner of the rectangle.

**4) Decryption:**

Take two letters at a time from digraph.

In case 1: instead of going right, go left.

In case 2: instead of going top to bottom, go bottom to top.

Case 3 remains the same.

**Program (Source Code):**

#include <bits/stdc++.h>

using namespace std;

string encrypt(string plainText, string key){

    string encryptedText;

    //make a matrix of 5\*5 and fill it with '-'

    char matrix[5][5];

    for (int i=0;i<5;i++){

        for (int j=0;j<5;j++){

            matrix[i][j] = '-';

        }

    }

    //make a hash array for all alphabets

    int hashArray[26] = {0};

    //mark j in hash array, as we will not use it. (we use i instead of j)

    int temp = 'j' - 'a';

    hashArray[temp] = 1;

    //fill the matrix with key

    //when filling a character, mark it in the hash array so that we don't insert it again afterwards

    int keyPointer = 0;

    for (int i=0;i<5;i++){

        for (int j=0;j<5;j++){

            //to stop filling once whole key is inserted, we put condition

            while (keyPointer < key.length()){

                //check if key is repeated or not

                if (hashArray[int(key[keyPointer]) - int('a')] == 0){

                    matrix[i][j] = key[keyPointer];

                    hashArray[int(key[keyPointer]) - int('a')] = 1;

                    keyPointer++;

                    break;  //means break current iteration and do next one

                }

                else{

                    keyPointer++;

                }

            }

        }

    }

    //filling the rest of the matrix alphabet wise, with no character repeat

    string alphabet = "abcdefghijklmnopqrstuvwxyz";

    int alphabetPointer = 0;

    for (int i=0;i<5;i++){

        for (int j=0;j<5;j++){

            //to reach the unfilled values

            if (matrix[i][j] == '-'){

                while (alphabetPointer < alphabet.length()){

                    if (hashArray[int(alphabet[alphabetPointer]) - int('a')] == 0){

                        matrix[i][j] = alphabet[alphabetPointer];

                        hashArray[int(alphabet[alphabetPointer]) - int('a')] = 1;

                        alphabetPointer++;

                        break;

                    }

                    else{

                        alphabetPointer++;

                    }

                }

            }

        }

    }

    // //printing the matrix

    // cout<<"\n";

    // for (int i=0;i<5;i++){

    //     for (int j=0;j<5;j++){

    //         cout<<matrix[i][j]<<" ";

    //     }

    //     cout<<"\n";

    // }

    // cout<<"\n";

    //replacing all 'j' in plainText by 'i'

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

        if (plainText[i] == 'j'){

            plainText[i] = 'i';

        }

    }

    //making the plainText (adding the bogus letter 'x' wherever two letters are same in pair)

    string actualText;

    int noOfxAdded = 0;     //used later

    int plainTextPointer = 0;

    while ((plainTextPointer+1) < plainText.length()){

        if (plainText[plainTextPointer] != plainText[plainTextPointer+1]){

            actualText += plainText[plainTextPointer];

            actualText += plainText[plainTextPointer+1];

            plainTextPointer = plainTextPointer+2;

        }

        else{

            actualText += plainText[plainTextPointer];

            actualText += 'x';

            noOfxAdded++;

            plainTextPointer++;

        }

    }

    //checking if last character needs to be added or not (cuz it is not added in the above steps sometimes, due to range issuess)

    if ((plainText.length() + noOfxAdded) != actualText.length()){

        actualText += plainText[plainText.length()-1];

    }

    //adding a bogus 'z' if length is odd

    if (actualText.length() % 2 != 0){

        actualText += 'z';

    }

    //plainText is updated and ready to be encrypted

    //form the encrypted text by finding pairs in matrix and replacing them

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

        char a = actualText[i];

        char b = actualText[i+1];

        //position of actual plain text characters

        int row\_a=0;

        int col\_a=0;

        int row\_b=0;

        int col\_b=0;

        //encrypted characters

        char ch1;

        char ch2;

        for (int j=0;j<5;j++){

            for (int k=0;k<5;k++){

                if (matrix[j][k] == a){

                    row\_a = j;

                    col\_a = k;

                }

                else if (matrix[j][k] == b){

                    row\_b = j;

                    col\_b = k;

                }

            }

        }

        //if both characters are in same row

        if (row\_a == row\_b){

            col\_a = (col\_a + 1)%5;

            col\_b = (col\_b + 1)%5;

        }

        //if both characters are in same column

        else if (col\_a == col\_b){

            row\_a = (row\_a + 1)%5;

            row\_b = (row\_b + 1)%5;

        }

        //if both are in different rows and columns

        else{

            //just interchange the col values

            int tempCol = col\_a;

            col\_a = col\_b;

            col\_b = tempCol;

        }

        ch1 = matrix[row\_a][col\_a];

        ch2 = matrix[row\_b][col\_b];

        encryptedText += ch1;

        encryptedText += ch2;

    }

    return encryptedText;

}

//similar but just opposite of encryption method.

//form the key matrix which will be the same.

//difference is only in forming the decryptedText.

string decrypt(string encryptedText, string key){

    string decryptedText;

    //make a matrix of 5\*5 and fill it with '-'

    char matrix[5][5];

    for (int i=0;i<5;i++){

        for (int j=0;j<5;j++){

            matrix[i][j] = '-';

        }

    }

    //make a hash array for all alphabets

    int hashArray[26] = {0};

    //mark j in hash array, as we will not use it. (we use i instead of j)

    int temp = 'j' - 'a';

    hashArray[temp] = 1;

    //fill the matrix with key

    //when filling a character, mark it in the hash array so that we don't insert it again afterwards

    int keyPointer = 0;

    for (int i=0;i<5;i++){

        for (int j=0;j<5;j++){

            //to stop filling once whole key is inserted, we put condition

            while (keyPointer < key.length()){

                //check if key is repeated or not

                if (hashArray[int(key[keyPointer]) - int('a')] == 0){

                    matrix[i][j] = key[keyPointer];

                    hashArray[int(key[keyPointer]) - int('a')] = 1;

                    keyPointer++;

                    break;  //means break current iteration and do next one

                }

                else{

                    keyPointer++;

                }

            }

        }

    }

    //filling the rest of the matrix alphabet wise, with no character repeat

    string alphabet = "abcdefghijklmnopqrstuvwxyz";

    int alphabetPointer = 0;

    for (int i=0;i<5;i++){

        for (int j=0;j<5;j++){

            //to reach the unfilled values

            if (matrix[i][j] == '-'){

                while (alphabetPointer < alphabet.length()){

                    if (hashArray[int(alphabet[alphabetPointer]) - int('a')] == 0){

                        matrix[i][j] = alphabet[alphabetPointer];

                        hashArray[int(alphabet[alphabetPointer]) - int('a')] = 1;

                        alphabetPointer++;

                        break;

                    }

                    else{

                        alphabetPointer++;

                    }

                }

            }

        }

    }

    //length of encryptedText (cipher text) is always even

    //form the decrypted text by finding pairs in matrix and replacing them

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

        char a = encryptedText[i];

        char b = encryptedText[i+1];

        //position of encryptedText characters

        int row\_a=0;

        int col\_a=0;

        int row\_b=0;

        int col\_b=0;

        //decrypted characters

        char ch1;

        char ch2;

        for (int j=0;j<5;j++){

            for (int k=0;k<5;k++){

                if (matrix[j][k] == a){

                    row\_a = j;

                    col\_a = k;

                }

                else if (matrix[j][k] == b){

                    row\_b = j;

                    col\_b = k;

                }

            }

        }

        //if both characters are in same row

        if (row\_a == row\_b){

            col\_a = (col\_a - 1 +5)%5;

            col\_b = (col\_b - 1 +5)%5;

        }

        //if both characters are in same column

        else if (col\_a == col\_b){

            row\_a = (row\_a - 1 +5)%5;

            row\_b = (row\_b - 1 +5)%5;

        }

        //if both are in different rows and columns

        else{

            //just interchange the col values

            int tempCol = col\_a;

            col\_a = col\_b;

            col\_b = tempCol;

        }

        ch1 = matrix[row\_a][col\_a];

        ch2 = matrix[row\_b][col\_b];

        decryptedText += ch1;

        decryptedText += ch2;

    }

    return decryptedText;

}

int main(){

    string plainText0 = "mynameisravi";

    string keyword0 = "pdeu";

    string plainText;

    string keyword;

    //truncate spaces from key and plainText, and convert to lowercase

    for (int i=0;i<plainText0.length();i++){

        if (plainText0[i] != ' '){

            plainText += tolower(plainText0[i]);

        }

    }

    for (int i=0;i<keyword0.length();i++){

        if (keyword0[i] != ' '){

            keyword += tolower(keyword0[i]);

        }

    }

    //replacing 'j' in keyword with 'i'

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

        if (keyword[i] == 'j'){

            keyword[i] = 'i';

        }

    }

    cout<<"Plain Text: "<<plainText;

    cout<<"\nKeyword: "<<keyword;

    string encryptedText = encrypt(plainText, keyword);

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

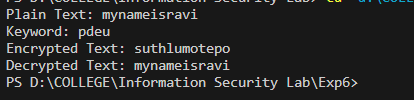
    string decryptedText = decrypt(encryptedText, keyword);

    cout<<"\nDecrypted Text: "<<decryptedText;

    return 0;

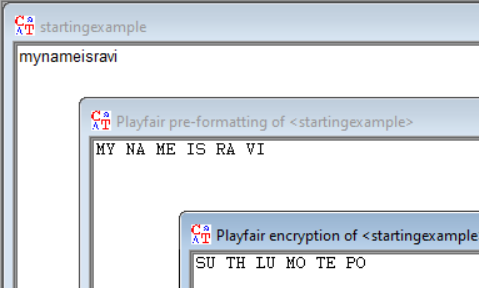
}

**Output (Program):**

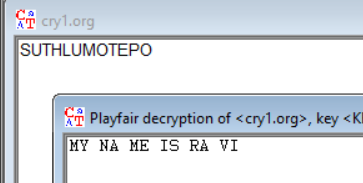
****

**Output (Cryptool):**

**Encryption:**

****

**Decryption:**



**Cryptanalysis:**

**Digraph Frequency Analysis:**

To perform digraph frequency analysis, begin by constructing a list of the most prevalent pairs of letters (digraphs) found in English text. Next, tally the occurrences of these digraphs in the encrypted message. If the distribution of digraph frequencies in the encrypted message differs from that in standard English text, this divergence can be utilized to deduce the encryption key. For instance, if the digraph "TH" appears more often in the encrypted message than it typically does in English text, it indicates that the key likely incorporates the "TH" digraph.

**Brute Force Attack:**

A brute force attack on the Playfair cipher involves systematically trying all possible 5x5 grids as decryption keys, using the 25 letters of the alphabet (minus duplicates) to form these grids. For each generated grid, the attacker decrypts the ciphertext and checks if the resulting text makes sense. This process continues until a valid decryption is found or all possibilities are exhausted. While it is time-consuming due to the large number of potential keys, it can eventually reveal the correct decryption key.

**Applications:**

• Military Communication

• Historical Encryption

• Classic Cryptography Education

**References:**

1. GeeksforGeeks
2. www.nku.edu
3. en.wikipedia.org

**Experiment-7**

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

**Introduction:**

In classical cryptography, the hill cipher is a polygraphic substitution cipher based on Linear Algebra. It was invented by Lester S. Hill in the year 1929. In simple words, it is a cryptography algorithm used to encrypt and decrypt data for the purpose of data security.

The algorithm uses matrix calculations used in Linear Algebra. It is easier to understand if we have the basic knowledge of matrix multiplication, modulo calculation, and the inverse calculation of matrices.

In hill cipher algorithm every letter (A-Z) is represented by a number moduli 26. Usually, the simple substitution scheme is used where A = 0, B = 1, C = 2…Z = 25 to use 2x2 key matrix.

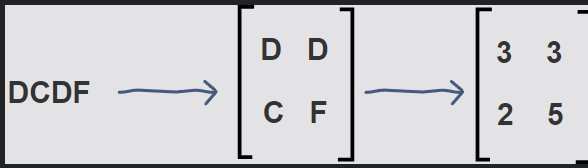
**Encryption:**

Encrypting with the Hill cipher is built on the following operation:

**E(K, P) = (K\*P) mod 26**

Where K is our key matrix and P is the plaintext in vector form. Matrix multiplying these two terms produces the encrypted ciphertext. Let us do so step by step:

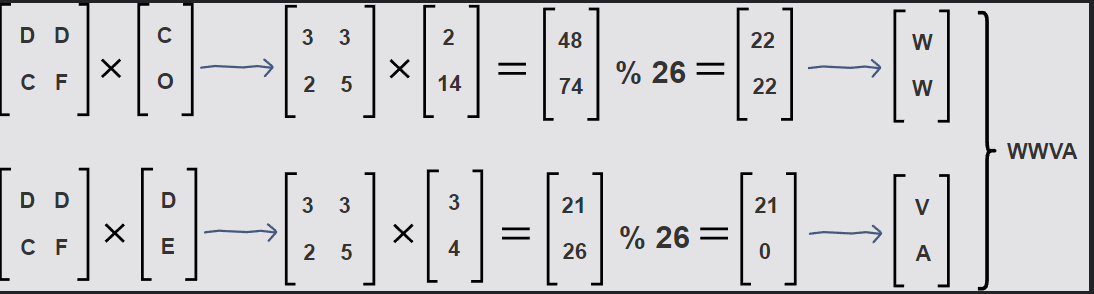
1. Pick a keyword to encrypt your plaintext message. Let us work with the random keyword “DDCF.” Convert this keyword to matrix form using your substitution scheme to convert it to a numerical 2x2 key matrix.



1. Next, we will convert our plaintext message to vector form. Since our key matrix is 2x2, the vector needs to be 2x1 for matrix multiplication to be possible. In our case, our message is four letters long so we can split it into blocks of two and then substitute to get our plaintext vectors.



1. Now, we can matrix multiply the key matrix with each 2x1 plaintext vector, take the moduli of the resulting 2x1 vectors by 26, and concatenate the results to get “WWVA”, the final ciphertext.



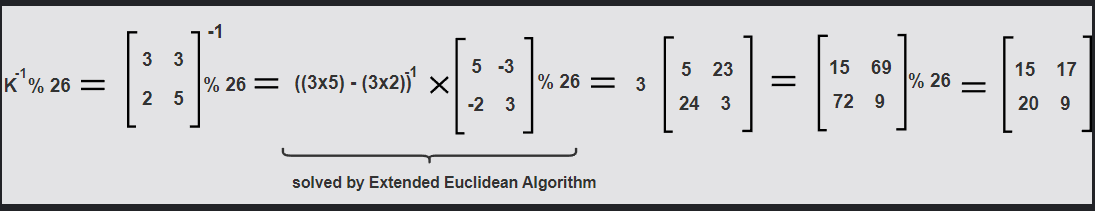
**Decryption:**

Decrypting with the Hill cipher is built on the following operation:

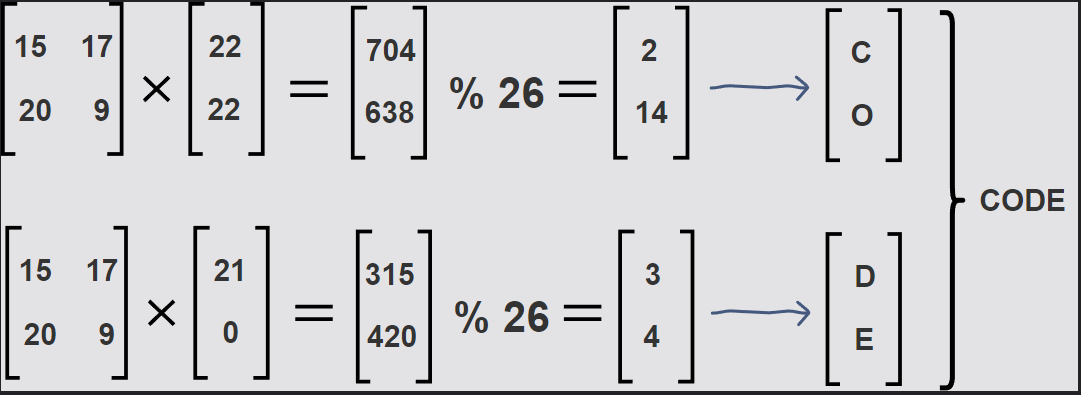
**D(K, C) = (K-1 \*C) mod 26**

Where K is our key matrix and C is the ciphertext in vector form. Matrix multiplying the inverse of the key matrix with the ciphertext produces the decrypted plaintext. Let us do this step by step with our ciphertext, "WWVA":

1. First, we calculate the inverse of the key matrix. In doing so, we must keep​ the result between 0-25 using modulo 26. For this reason, the Extended Euclidean algorithm is used to find the modular multiplicative inverse of the key matrix determinant.



1. Next, we will multiply 2x1 blocks of the ciphertext with the inverse of the key matrix to get our original plaintext message, “CODE,” back.



**Program (Source Code):**

#include <bits/stdc++.h>

using namespace std;

void performEncryption()

{

    int x, y, i, j, k, n;

    cout << "Enter the encryption key: ";

    string key;

    cin >> key;

    n = sqrt(key.length());

    vector<vector<int>> matrix(n, vector<int>(n));

    int iter = 0;

    for (i = 0; i < n; i++)

    {

        for (j = 0; j < n; j++)

        {

            int val = key[iter] - 97;

            matrix[i][j] = val;

            iter++;

        }

    }

    cout << "Key matrix:" << endl;

    for (i = 0; i < n; i++)

    {

        for (j = 0; j < n; j++)

        {

            cout << matrix[i][j] << " ";

        }

        cout << endl;

    }

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

    string message;

    cin >> message;

    int padding = (n - message.size() % n) % n;

    for (i = 0; i < padding; i++)

    {

        message += 'x';

    }

    k = 0;

    string encryptedText = "";

    while (k < message.size())

    {

        for (i = 0; i < n; i++)

        {

            int sum = 0;

            int temp = k;

            for (j = 0; j < n; j++)

            {

                sum += (matrix[i][j] % 26 \* (message[temp++] - 'a') % 26) % 26;

                sum = sum % 26;

            }

            encryptedText += (sum + 'a');

        }

        k += n;

    }

    cout << "\nEncrypted text is: " << encryptedText << '\n';

}

int modInverse(int a, int m)

{

    a = a % m;

    for (int x = -m; x < m; x++)

        if ((a \* x) % m == 1)

            return x;

    return 0;

}

void getCofactor(vector<vector<int>> &a, vector<vector<int>> &temp, int p, int q, int n)

{

    int i = 0, j = 0;

    for (int row = 0; row < n; row++)

    {

        for (int col = 0; col < n; col++)

        {

            if (row != p && col != q)

            {

                temp[i][j++] = a[row][col];

                if (j == n - 1)

                {

                    j = 0;

                    i++;

                }

            }

        }

    }

}

int calculateDeterminant(vector<vector<int>> &a, int n, int N)

{

    int D = 0;

    if (n == 1)

        return a[0][0];

    vector<vector<int>> temp(N, vector<int>(N));

    int sign = 1;

    for (int f = 0; f < n; f++)

    {

        getCofactor(a, temp, 0, f, n);

        D += sign \* a[0][f] \* calculateDeterminant(temp, n - 1, N);

        sign = -sign;

    }

    return D;

}

void getAdjoint(vector<vector<int>> &a, vector<vector<int>> &adj, int N)

{

    if (N == 1)

    {

        adj[0][0] = 1;

        return;

    }

    int sign = 1;

    vector<vector<int>> temp(N, vector<int>(N));

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            getCofactor(a, temp, i, j, N);

            sign = ((i + j) % 2 == 0) ? 1 : -1;

            adj[j][i] = (sign) \* (calculateDeterminant(temp, N - 1, N));

        }

    }

}

bool calculateInverse(vector<vector<int>> &a, vector<vector<int>> &inv, int N)

{

    int det = calculateDeterminant(a, N, N);

    if (det == 0)

    {

        cout << "Inverse does not exist";

        return false;

    }

    int invDet = modInverse(det, 26);

    cout << det % 26 << ' ' << invDet << '\n';

    vector<vector<int>> adj(N, vector<int>(N));

    getAdjoint(a, adj, N);

    for (int i = 0; i < N; i++)

        for (int j = 0; j < N; j++)

            inv[i][j] = (adj[i][j] \* invDet) % 26;

    return true;

}

void performDecryption()

{

    int x, y, i, j, k, n;

    cout << "Enter the decryption key: ";

    string key;

    cin >> key;

    n = key.length() / 2;

    vector<vector<int>> a(n, vector<int>(n));

    vector<vector<int>> adj(n, vector<int>(n));

    vector<vector<int>> inv(n, vector<int>(n));

    int iter = 0;

    for (i = 0; i < n; i++)

    {

        for (j = 0; j < n; j++)

        {

            int val = key[iter] - 97;

            a[i][j] = val;

            iter++;

        }

    }

    if (calculateInverse(a, inv, n))

    {

        cout << "Inverse exists\n";

    }

    cout << "Enter the message to decrypt\n";

    string message;

    cin >> message;

    k = 0;

    string decryptedText;

    while (k < message.size())

    {

        for (i = 0; i < n; i++)

        {

            int sum = 0;

            int temp = k;

            for (j = 0; j < n; j++)

            {

                sum += ((inv[i][j] + 26) % 26 \* (message[temp++] - 'a') % 26) % 26;

                sum = sum % 26;

            }

            decryptedText += (sum + 'a');

        }

        k += n;

    }

    int lastCharIndex = decryptedText.size() - 1;

    while (decryptedText[lastCharIndex] == 'x')

    {

        lastCharIndex--;

    }

    cout << "\nDecrypted text is: ";

    for (i = 0; i <= lastCharIndex; i++)

    {

        cout << decryptedText[i];

    }

    cout << '\n';

}

int main()

{

    int choice;

    cout << "Enter your choice :\n";

    cout << "1. Encryption:\n2.Decryption:\n";

    cin >> choice;

    switch (choice)

    {

    case 1:

        performEncryption();

        break;

    case 2:

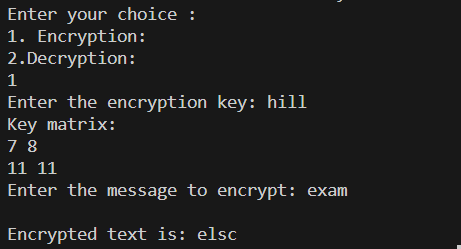
        performDecryption();

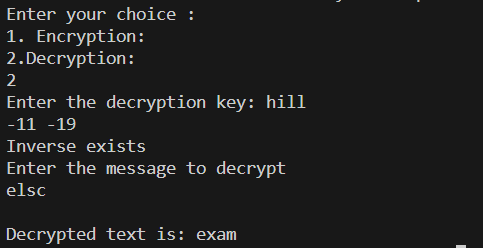
        break;

    }

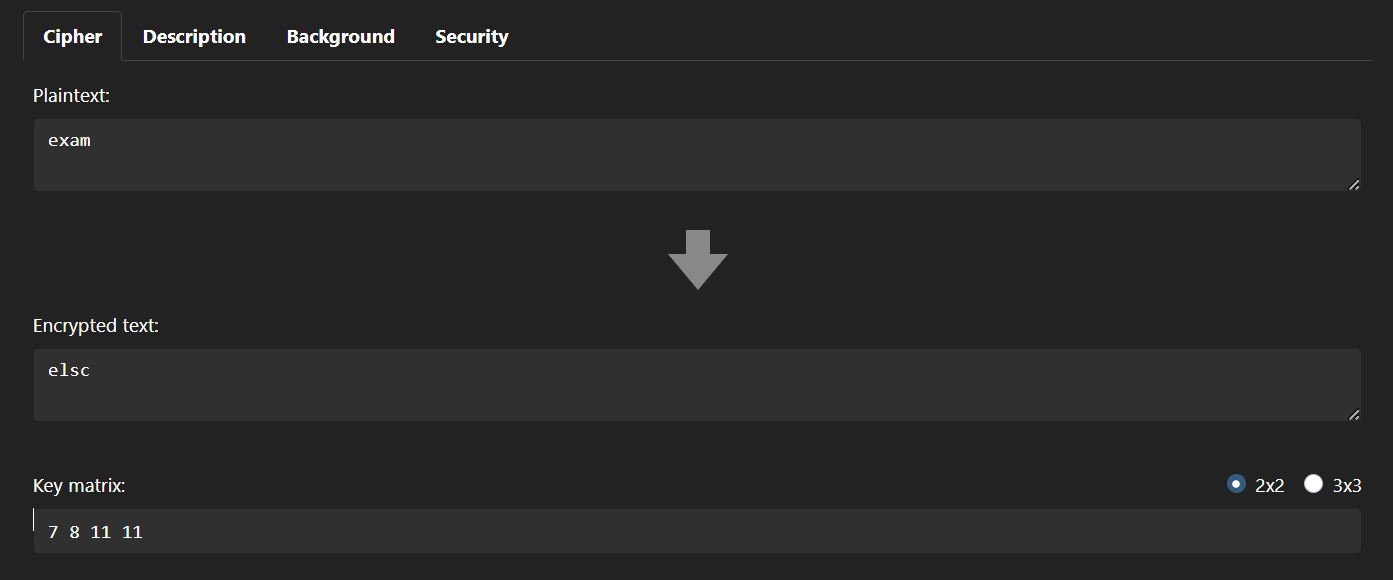
}

**Output (Program):**





**Output (Cryptool):**



**Cryptanalysis:**

**1) Known Plaintext Attack:**

In a known-plaintext attack, the attacker has access to some examples of plaintext and their corresponding ciphertext pairs. This information helps in deducing the key.

**2) Meet-in-the-Middle Attack:**

This attack involves encrypting all possible plaintexts using half of the key space, and decrypting all possible ciphertexts using the other half. If a match is found between the two sets, the key can be recovered.

**3) Linear Algebra:** The Hill cipher is based on matrix multiplication. With enough known plaintext-ciphertext pairs, the attacker can set up a system of linear equations to solve for the key matrix. The attacker needs at least as many pairs as the size of the key matrix (e.g., 4 pairs for a 2x2 matrix).

**Applications:**

While the Hill Cipher has certain vulnerabilities, it still finds application in various domains due to its matrix-based encryption. Some applications include:

**Educational Purposes:** The Hill Cipher is often used as an introductory example of a polygraphic cipher that utilizes matrix operations. It helps students learn about encryption, linear algebra, and modular arithmetic.

**Historical Significance:** The Hill Cipher is historically significant as one of the earliest attempts to enhance the security of classical ciphers. It paved the way for more advanced encryption techniques, including modern block ciphers.

**Secure Communication Protocols:** While not suitable for modern cryptographic standards, Hill Cipher's matrix-based approach can inspire more complex encryption algorithms used in secure communication protocols.

**Basic Encryption:** In scenarios where moderate security is sufficient, the Hill Cipher can be used for basic encryption of small texts, especially when education or historical context is the primary objective.

**References:**

* 1. GeeksforGeeks
  2. www.javatpoint.com
  3. www.educative.io
  4. www.cryptool.org/en/cto/hill

**Experiment-8**

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

**Introduction:**

The realm of cryptography holds a crucial role in ensuring data security and confidentiality across various contexts, including secure communication, data storage, and authentication. Cryptographic algorithms are pivotal in safeguarding sensitive information from unauthorized access. In this study, we investigate the utilization of encryption and decryption techniques with the Crypto++ library, a popular C++ library dedicated to cryptographic operations. Our primary focus centers on examining a range of block ciphers, which are symmetric key algorithms utilized for both data encryption and decryption.

**Block Ciphers:**

Block ciphers are a category of symmetric key ciphers designed to process fixed-size data blocks, typically of sizes such as 128, 192, or 256 bits. These ciphers employ a single key for both the encryption and decryption processes, making them suitable for scenarios where data security relies on a shared secret key. Examples of widely used block ciphers encompass the Advanced Encryption Standard (AES), Data Encryption Standard (DES), and Triple DES (3DES). Each of these ciphers employs distinct encryption and decryption techniques, rendering them suitable for diverse use cases.

**The Crypto++ Library:**

Crypto++ stands as a robust and versatile C++ library that delivers a variety of cryptographic algorithm implementations, including block ciphers. It provides a standardized interface for encryption and decryption operations, making it a valuable tool for ensuring secure data handling. This library comprises classes and functions tailored for AES, DES, 3DES, and numerous other encryption algorithms, facilitating seamless integration of cryptographic functionality into software applications.

**AES Encryption and Decryption:**

The Advanced Encryption Standard (AES) ranks among the most widely adopted block ciphers. AES operates on fixed 128-bit data blocks and supports key sizes of 128, 192, and 256 bits. It adopts a substitution-permutation network (SPN) structure, which encompasses substitution, permutation, and key mixing layers to deliver robust encryption. In our experiment, we will illustrate the process of implementing AES encryption and decryption using the Crypto++ library. AES is renowned for its security, efficiency, and extensive application across various domains.

**DES and 3DES Encryption and Decryption:**

The Data Encryption Standard (DES) and Triple DES (3DES) represent older block ciphers that have found extensive usage in the past. DES functions on 64-bit data blocks and utilizes a 56-bit key. 3DES represents an enhancement of DES, bolstering security by applying the DES algorithm thrice consecutively. In our investigation, we will delve into the implementation of both DES and 3DES encryption and decryption using Crypto++. Although these ciphers are regarded as legacy due to their relatively smaller key sizes, they remain relevant in specific applications.

**Program (Source 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;

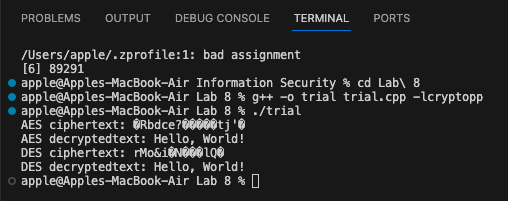
std::cout << "DES decryptedtext: " << decryptedtext << std::endl;

}

return 0;

}

**Output (Program):**



**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 for these ciphers:

**1) Brute Force Attacks:** Brute force attacks involve trying 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. For AES, the key length options are 128, 192, and 256 bits, with longer keys providing stronger security.

**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 selecting plaintexts for encryption. The goal is to deduce information about the key or the cipher's internal structure.

**3) Differential and Linear Cryptanalysis:** These are sophisticated techniques used to analyse the behaviour of block ciphers in relation to plaintext and ciphertext differences. Differential cryptanalysis focuses on the differences between pairs of plaintexts and how they affect the ciphertext, potentially revealing patterns in the cipher's operation. Linear cryptanalysis looks for linear relationships between the bits of the plaintext, ciphertext, and key. Both techniques can uncover biases and patterns that might 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. Successful attacks on the key schedule can lead to a complete compromise of the encryption. In particular, the design and security of the key schedule are critical in the strength of a block cipher.

**5) Side-Channel Attacks:** Cryptanalysis also encompasses side-channel attacks, which exploit information leaked during the encryption process, such as power consumption, electromagnetic radiation, or execution time. Implementations of block ciphers must be resistant to such attacks. Protecting against side-channel attacks is as important as the theoretical strength of the cipher itself.

**6) Block Cipher Modes of Operation:** Cryptanalysis can focus on the modes of operation used with block ciphers, such as Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), and Output Feedback (OFB). Understanding their security properties and potential weaknesses is essential, as improper usage of these modes can lead to vulnerabilities in the overall encryption scheme.

**Applications:**

**1) Data Encryption:** AES, DES, and 3DES are widely used to encrypt sensitive data. They play a vital role in securing data at rest, in transit, and in storage. This is essential for protecting confidential information in various contexts, including financial transactions, healthcare records, and sensitive communication.

**2) Virtual Private Networks (VPNs):** These ciphers are utilized in VPNs to establish secure and encrypted connections over untrusted networks, such as the internet. This ensures the confidentiality and integrity of data transmitted between remote locations and corporate networks.

**3) Secure Communication:** AES, DES, and 3DES are employed in secure communication protocols, such as HTTPS for web browsing and SSH for secure remote access. They protect the confidentiality of data exchanged between clients and servers, preventing eavesdropping and man-in-the-middle attacks.

**4) Wireless Network Security:** In wireless networks, like Wi-Fi, these ciphers are used to encrypt data to prevent unauthorized access. For instance, WPA and WPA2, which are common Wi-Fi security protocols, incorporate AES for encryption.

**5) Data Storage Encryption:** They are employed to encrypt data stored on various devices, including hard drives and flash drives. Full-disk encryption and file-level encryption solutions utilize these ciphers to safeguard data from unauthorized access in case of theft or loss.

**6) Secure File Transfer:** These ciphers are essential for secure file transfer protocols like SFTP (SSH File Transfer Protocol) and secure email communications. They ensure that files and messages remain confidential during transit.

**7) Financial Transactions:** In the financial sector, these ciphers are used to secure online banking transactions and payment processing systems, safeguarding the confidentiality of financial data.

**8) Database Encryption:** AES, DES, and 3DES are used to encrypt sensitive data in databases, ensuring that even if the database is compromised, the data remains protected.

**9) Smart Card and Chip Security:** These ciphers are used in secure smart cards, SIM cards, and integrated circuit chips to protect sensitive information like personal identification and payment data.

**10) Government and Military Applications:** These ciphers are used by government agencies and the military to protect classified and sensitive information, including communications, intelligence data, and more.

**11) IoT (Internet of Things) Security:** With the growing number of IoT devices, AES, DES, and 3DES are used to secure communication between IoT devices and networks, preventing unauthorized access and data breaches.

These ciphers have a wide range of applications, and their usage is determined by the specific security requirements of each application and the level of security provided by each cipher. It is worth noting that while AES is considered highly secure and is the preferred choice for most applications today, DES and 3DES are less commonly used due to their vulnerabilities and are often being phased out in favour of more modern encryption algorithms.

**References:**

1. https://cryptopp.com

**Experiment-9**

**Aim:** Study and implement RSA Encryption and Decryption function.

**Introduction:**

RSA, which stands for Rivest–Shamir–Adleman, is a popular method for keeping our digital messages and information safe when they travel over the internet.

RSA algorithm is an asymmetric cryptography algorithm. Asymmetric means that it works on two different keys i.e., Public Key and Private Key. As the name describes that the Public Key is given to everyone and the Private key is kept private.

**RSA is like a special lock and key system for your digital messages. Here is how it works:**

1. You use the recipient's public key to lock up your message, making it super secure.
2. The recipient uses their private key to unlock and read the message.

It helps uskeep our messages safe before sending them. And ensure that the message has not been tampered with during its journey.

RSA was the first successful way to use these special keys in the digital world.

**Method to generate public and private keys:**

* We take 2 prime numbers p & q.
* Multiply these numbers to find n = p x q, where n is called the modulus for encryption and decryption.
* Choose a number **e** less than **n**, such that n is relatively prime to **(p - 1) x (q -1).** It means that **e** and **(p - 1) x (q - 1)** have no common factor except 1. Choose "e" such that 1<e < φ (n), e is prime to φ (n),  
  **gcd (e,d(n)) =1**
* If **n = p x q,** then the public key is <e, n>. A plaintext message **m** is encrypted using public key <e, n>. To find ciphertext from the plain text following formula is used to get ciphertext C.

**C = me mod n**  
Here**, m** must be less than **n**. A larger message (>n) is treated as a concatenation of messages, each of which is encrypted separately.

* To determine the private key, we use the following formula to calculate the d such that:

**d\*e mod {(p - 1) x (q - 1)} = 1**  
**Or**  
**d\*e mod φ (n) = 1**

* The private key is <d, n>. A ciphertext message **c** is decrypted using private key <d, n>. To calculate plain text **m** from the ciphertext c following formula is used to get plain text m.

**m = cd mod n**

**Program (Source Code):**

#include <bits/stdc++.h>

using namespace std;

int T1 = 0;

int T2 = 1;

int gcd(int a, int b) {

    int temp;

    while (1){

        temp = a % b;

        if (temp == 0){

            return b;

        }

        a = b;

        b = temp;

    }

}

int extendedEuclidean(int a, int b){

    // a > b

    int A = a;

    int B = b;

    int Q = A/B;

    int R = A%B;

    int T = T1 - (T2\*Q);

    // cout<<Q<<" "<<A<<" "<<B<<" "<<R<<" "<<T1<<" "<<T2<<" "<<T<<" "<<endl;

    A = B;

    B = R;

    T1 = T2;

    T2 = T;

    if (B == 0){

        return T1;

    }

    else{

        return extendedEuclidean(A, B);

    }

}

double powerMod(int plaintext, double power, double n){

    if (int(power) == 0) {

        return 1;

    }

    double result = powerMod(plaintext, power / 2, n);

    result = int(result \* result) % int(n);

    if (int(power) % 2 == 1) {

        result = int(result \* plaintext) % int(n);

    }

    return result;

}

int main() {

    double p = 3;

    double q = 7;

    double n = p \* q;

    double phi = (p - 1) \* (q - 1);

    double e = 2;

    // calculating value of e

    while (e < phi){

        // e must be co-prime to phi and

        // smaller than phi.

        if (gcd(e, phi) == 1)

            break;

        else

            e++;

    }

    // calculating value of d

    double d = extendedEuclidean(phi, e);

    while (d<0){

        d += phi;

    }

    // cout<<"e: "<<e<<endl;

    // cout<<"d: "<<d<<endl;

    // cout<<"n: "<<n<<endl;

    int plaintext = 12;

    cout<<"Plaintext message: "<<plaintext<<endl;

    double cipherText = powerMod(plaintext,e,n);

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

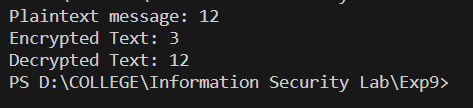
    double decryptedText = powerMod(cipherText,d,n);

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

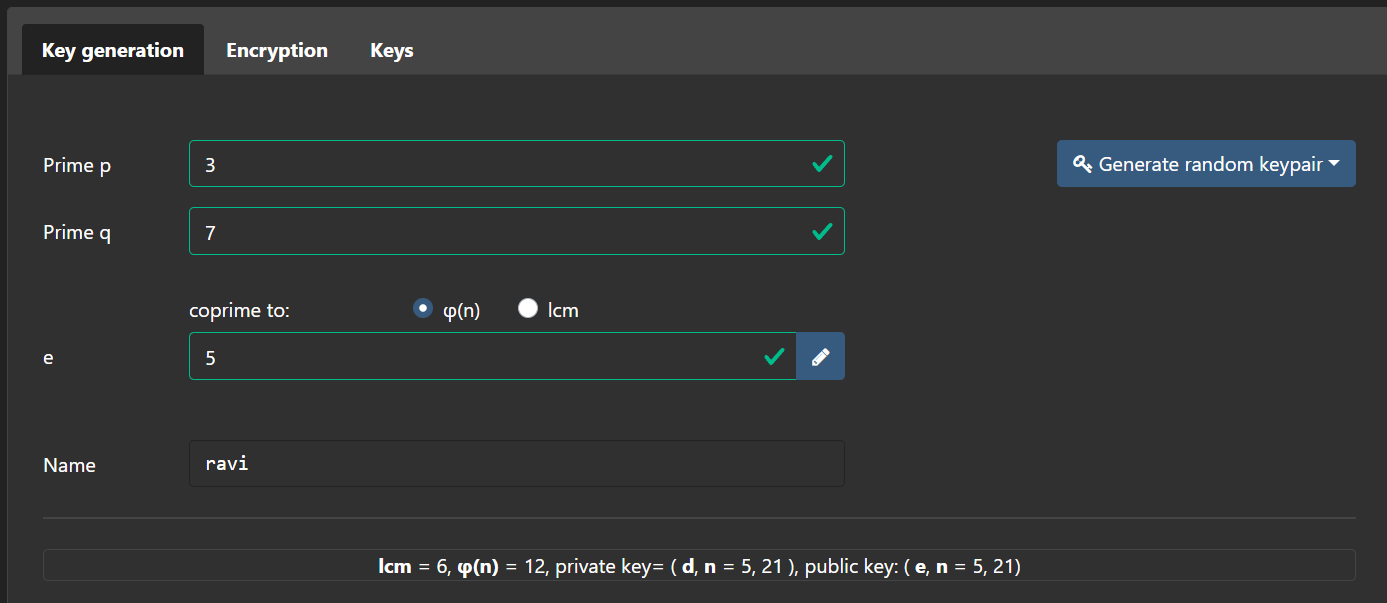
    return 0;

}

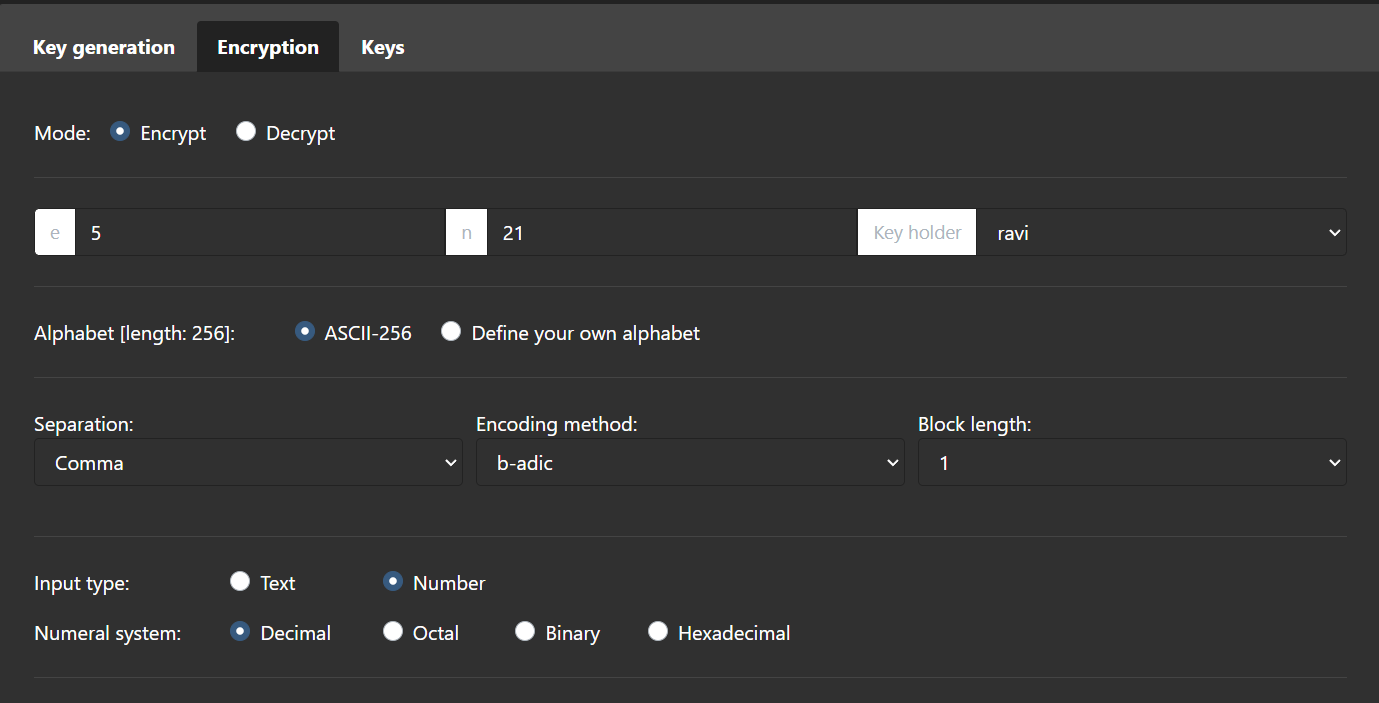
**Output (Program):**

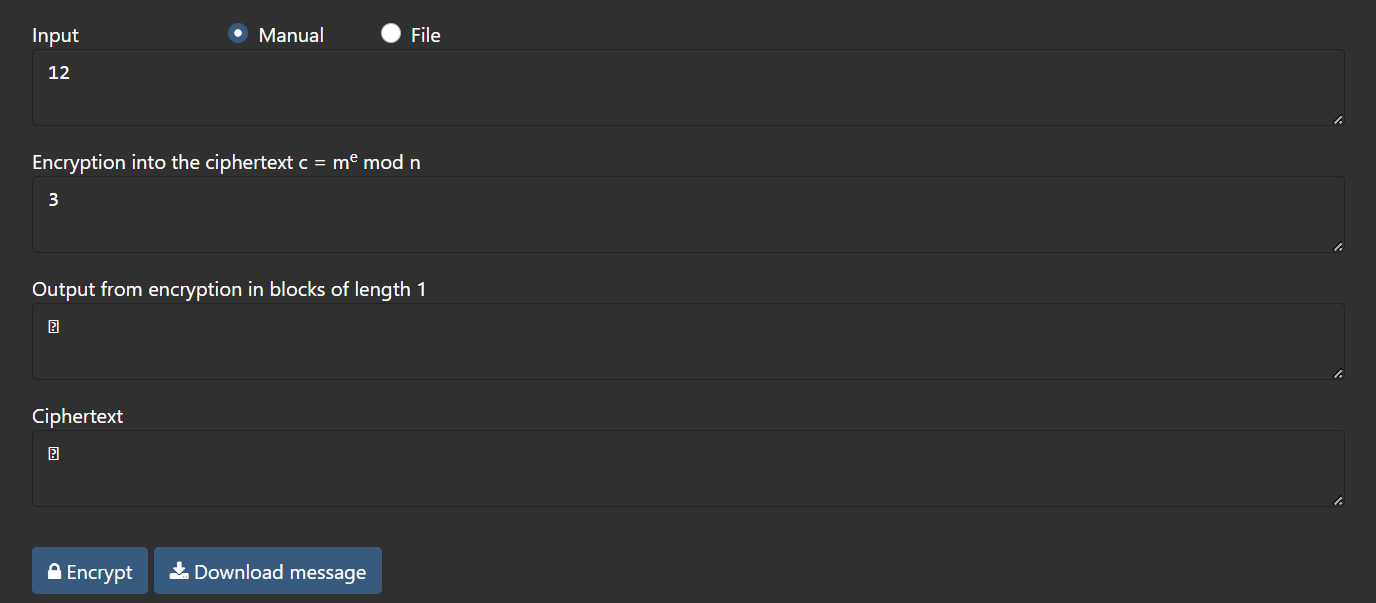
****

**Output (Cryptool):**

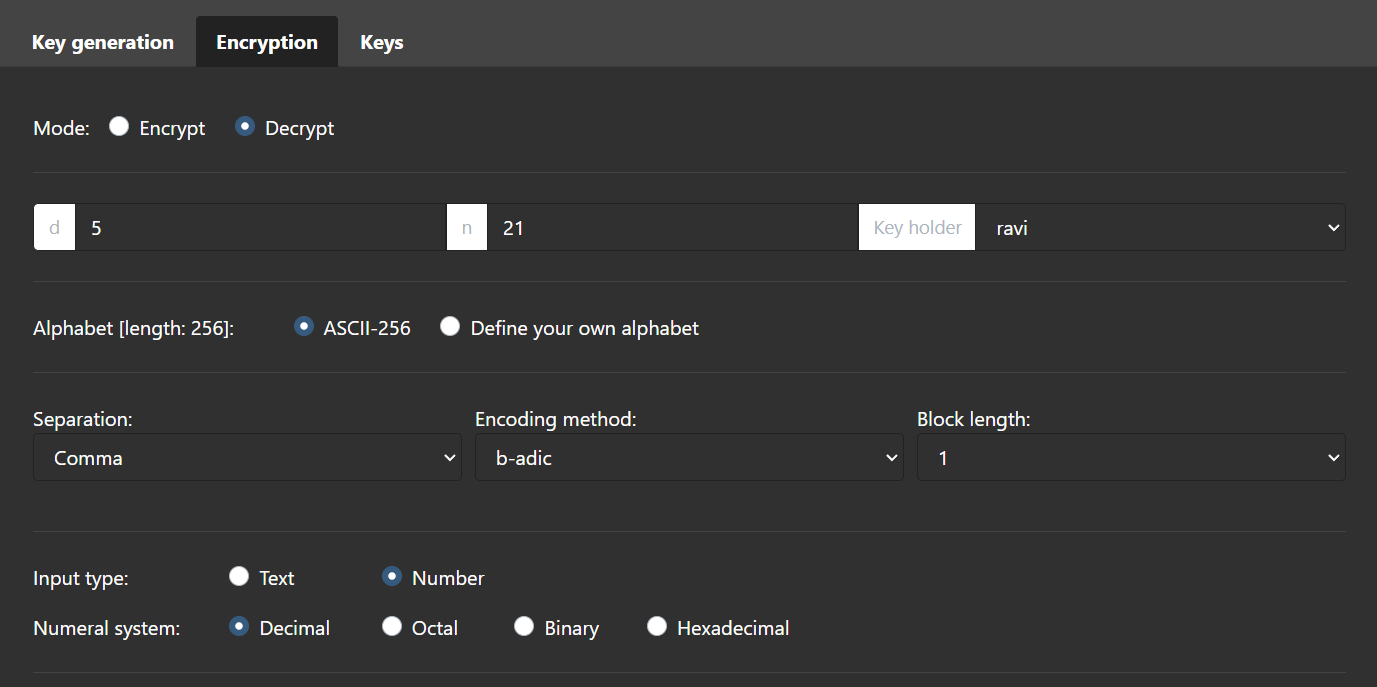
****

Encryption:

****

****

Decryption:





**Cryptanalysis:**

**Timing Attacks**: Think of this like spying on a clock. Attackers watch how long it takes to open a locked message. It is hard to do, and they need super-precise measurements to learn anything.

**Quantum Computing**: Future quantum computers, if sufficiently powerful, could potentially factor large numbers significantly faster than classical computers using Shor's algorithm. This represents a potential threat to RSA encryption.

**Common Modulus Attack**: In some cases where multiple parties use the same modulus (n) with different public exponents (e), a common modulus attack can be mounted to recover the shared modulus and, subsequently, the private keys.

**Applications:**

RSA (Rivest-Shamir-Adleman) encryption is widely used for:

**1. Secure Communication:** Protects data during transmission, like online banking and e-commerce transactions.

**2. SSL/TLS Encryption:** Fundamental for secure web browsing.

**3. Digital Signatures:** Verifies document authenticity and integrity.

**4. Secure Messaging:** Encrypts messages in various apps.

**5. Secure Email:** Used in PGP and S/MIME for email encryption.

**6. Authentication:** Enables secure login, including 2FA and SSH.

**7. VPN Security:** Establishes secure internet tunnels.

**8. IoT Security:** Used in securing IoT device communication.

**9. Smart Cards and Tokens:** Secures private keys for secure access.

**10. Digital Certificates:** Validates website authenticity.

**11. Secure File Storage:** Encrypts data on disks and cloud storage.

**12. Secure Backup:** Ensures backup data is protected.

RSA remains vital for safeguarding digital communication and data.

**References:**

* 1. GeeksforGeeks
  2. www.cryptool.org/en/cto/hill

**Experiment-10**

**Aim:** Use RSA for generation and verification of digital signature on file.

**Introduction:**

RSA digital signatures are like a digital seal of approval. They make sure that online stuff is real and has not been messed with. Here is how they work:

There are two keys, one for locking (public) and one for unlocking (private).

The private key makes a unique signature for the data, like a special stamp.

Other people can use the public key to check if the stamp is real, making sure nobody can deny sending it.

RSA works because it is hard to crack the code used for the stamp, and it is crucial for trust and safety online, like making sure documents are real and online payments are secure.

**Program (Source Code):**

// senders's authentication only

#include <bits/stdc++.h>

#include <cmath>

double PrivateP = 13;

double PrivateQ = 23;

double PrivateN = PrivateP \* PrivateQ;

double phi = (PrivateP - 1) \* (PrivateQ - 1);

double PrivateE = 2;

int T1 = 0;

int T2 = 1;

using namespace std;

double powerMod(int plaintext, double power, double n){

    if (int(power) == 0) {

        return 1;

    }

    double result = powerMod(plaintext, power / 2, n);

    result = int(result \* result) % int(n);

    if (int(power) % 2 == 1) {

        result = int(result \* plaintext) % int(n);

    }

    return result;

}

int gcd(int a, int b) {

    int temp;

    while (1){

        temp = a % b;

        if (temp == 0){

            return b;

        }

        a = b;

        b = temp;

    }

}

// to calculate inverse of a number

// to calculate value of d (inverse of e) modulo n

int extendedEuclidean(int a, int b){

    // a > b

    int A = a;

    int B = b;

    int Q = A/B;

    int R = A%B;

    int T = T1 - (T2\*Q);

    // cout<<Q<<" "<<A<<" "<<B<<" "<<R<<" "<<T1<<" "<<T2<<" "<<T<<" "<<endl;

    A = B;

    B = R;

    T1 = T2;

    T2 = T;

    if (B == 0){

        return T1;

    }

    else{

        return extendedEuclidean(A, B);

    }

}

int calculateE(int PrivateE){

    // calculating value of PrivateE

    while (PrivateE < phi){

        // e must be co-prime to phi and

        // smaller than phi.

        if (gcd(PrivateE, phi) == 1){

            break;

        }

        else{

            PrivateE++;

        }

    }

    return PrivateE;

}

// to generate MD1 and MD2 for message m

int hashingFunction(string m){

    int hash = 0;

    for (char c : m){

        hash = (hash \* 31) + c;

    }

    return hash;

}

// to generate digital signature for a md value generated in hashingFunction()

int generateDigitalSignature(int md){

    int k = 2;

    double PrivateD = extendedEuclidean(phi, PrivateE);

    while (PrivateD < 0){

        PrivateD += phi;

    }

    // cout<<phi<<endl;

    // cout<<PrivateE<<endl;

    // cout<<PrivateD<<endl;

    // cout<<PrivateN<<endl;

    // cout<<power<<endl;

    // double power = pow((int)md, (int)(PrivateD));

    // int power2 = int(power);

    // double digitalSignature = power2 % int(PrivateN);

    double digitalSignature = powerMod(md, PrivateD, PrivateN);

    return digitalSignature;

}

int decryptDigitalSignature(int digitalSignature){

    double PublicE = PrivateE;

    double PublicN = PrivateN;

    // double power = pow((int)digitalSignature, (int)(PublicE));

    // int power2 = int(power);

    // int md = power2 % (int)PublicN;

    double md0 = powerMod(int(digitalSignature), PublicE, PublicN);

    int md = int(md0);

    return md;

}

int checkAuthenticity(int originalMessage, int digitalSignature){

    int md = decryptDigitalSignature(digitalSignature);

    if (originalMessage == md){

        return true;

    }

    else{

        return false;

    }

}

int main() {

    // to calculate value of e which is needed by both Alice and Bob (sender & receiver)

    PrivateE = calculateE(PrivateE);

    int originalMessage = 130;

    cout<<"Original message: "<<originalMessage<<endl;

    int digitalSignature = generateDigitalSignature(originalMessage);

    cout<<"Digital Signature generated: "<<digitalSignature<<endl;

    if (checkAuthenticity(originalMessage, digitalSignature)){

        cout<<"Authenication Successful!"<<endl;

    }

    else{

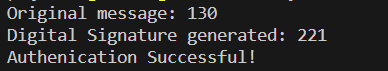
        cout<<"Authentication Failed!"<<endl;

    }

    return 0;

}

**Output (Program):**

****

**Cryptanalysis:**

**Advantages of RSA Technique:**

1. Security – RSA algorithm is a very secure method for encrypting and decrypting sensitive information. It uses the properties of large prime numbers to make it difficult for outsiders to crack the code.
2. Digital Signatures – RSA algorithm can be used to create digital signatures, which can help to verify the authenticity of digital documents.
3. Speed – RSA algorithm is relatively fast and efficient, making it suitable for use in real-time applications.

**Disadvantages of RSA Technique:**

1. Key Size – RSA algorithm requires large prime numbers as part of the encryption process. The larger the prime numbers, the more secure the encryption, but it also increases the key size and processing time.
2. Vulnerability to Quantum Computing – RSA algorithm is vulnerable to attacks by quantum computers, which can potentially break the encryption.

**Applications:**

**1) Secure Communication:** Digital signatures are used to ensure that messages or data exchanged between parties have not been tampered with during transmission. This is essential for secure email communication, instant messaging, and online chats.

**2) Software and Firmware Authentication:** Digital signatures are employed to verify the authenticity and integrity of software applications, updates, and firmware. This prevents the installation of malicious or tampered software.

**3) Document Authentication:** In legal and business contexts, digital signatures are used to verify the authenticity of electronic documents, contracts, and agreements. This eliminates the need for physical signatures and provides a secure method for electronic document management.

**4) E-commerce and Online Transactions:** Digital signatures play a crucial role in online payment systems and e-commerce. They ensure that online transactions are secure and that the transaction details are unaltered.

**References:**

1. GeeksforGeeks

2. www.javatpoint.com

3. www.cryptool.org/en/cto/hill