****

KARTIK PATEL

21BCP162

DIV-3/G-5

INFORMATION SECURITY LAB

**LAB FILE**

EXPERIMENT NO – 1

**Objective:** Download and Practice CrypTool

**Introduction:**

CrypTool is an open-source software project designed to facilitate comprehensive learning and experimentation in cryptography and cryptanalysis. It offers a valuable platform for students, researchers, and professionals seeking to grasp the intricacies of secure communication and data protection. CrypTool's main aim is to demystify the complex realm of cryptography, making it accessible even to those without a strong background in mathematics or computer science.

**Main Features of CrypTool:**

User-Friendly Interface: CrypTool presents an intuitive graphical user interface (GUI), enabling users to explore and experiment with diverse cryptographic algorithms and techniques without delving deeply into the mathematical underpinnings.

Educational Modules: CrypTool offers a variety of educational modules covering different cryptographic concepts, algorithms, and attack strategies. These modules provide step-by-step explanations, visual aids, and interactive tools to facilitate comprehension of encryption and decryption processes.

Cryptanalysis Tools: In addition to cryptography, CrypTool includes tools for cryptanalysis, which involves breaking cryptographic codes. Users can practice attacking and breaking various encryption schemes, gaining insights into the vulnerabilities of different algorithms.

Real-World Applications: CrypTool demonstrates practical uses of cryptography in real-world scenarios, enabling users to simulate encryption and decryption processes as they might be applied in secure communication, digital signatures, and other cryptographic protocols.

**Utilization Scenarios:**

**Education:** CrypTool is widely used in academic settings to teach cryptography and cryptanalysis, offering a hands-on approach that enhances understanding.

**Research:** Researchers utilize CrypTool to prototype and test new cryptographic algorithms, protocols, and attacks within a controlled environment.

Some of the Provided Ciphers by CrypTool:

**Historical Ciphers:**

CrypTool features historical ciphers such as the Caesar cipher, Vigenère cipher, Hill Cipher, and Playfair cipher. These ciphers, used in earlier times for encryption, serve educational purposes and aid in understanding the evolution of cryptography.

**Symmetric-Key Ciphers:**

AES (Advanced Encryption Standard): AES is a widely adopted symmetric-key block cipher renowned for robust security and efficient encryption/decryption processes, operating on fixed-size data blocks and supporting various key lengths.

DES (Data Encryption Standard): DES, an older symmetric-key block cipher, was once widespread but is now considered relatively insecure due to its small key size. It operates on 64-bit blocks using a 56-bit key.

Triple DES (3DES): 3DES is a DES variant that applies the DES algorithm thrice to each data block. While more secure than single DES, it is slower and less efficient than modern alternatives like AES.

Blowfish and Twofish: These symmetric-key block ciphers are designed for rapid encryption and decryption. Blowfish has a 64-bit block size, while Twofish offers greater flexibility in block size and key length.

**Asymmetric-Key Ciphers:**

RSA: RSA is a prevalent asymmetric-key cipher for secure communication and digital signatures, relying on the properties of large prime numbers and providing a public-private key pair for encryption and decryption.

ElGamal: ElGamal is another asymmetric-key cipher offering encryption and digital signatures, based on the challenge of solving the discrete logarithm problem within a finite field.

Elliptic Curve Cryptography (ECC): ECC comprises a family of asymmetric-key ciphers reliant on the algebraic structure of elliptic curves. It offers robust security with shorter key lengths compared to traditional ciphers like RSA.

**References:**

CrypTool's official website: [www.CrypTool.org](http://www.CrypTool.org)

Wikipedia page on CrypTool: <https://en.wikipedia.org/wiki/CrypTool>

EXPERIMENT NO – 2

**Aim:** The aim of this experiment is to explore the principles, implementation, cryptanalysis, applications, and historical significance of the Caesar cipher encryption and decryption algorithm.

**Introduction:** The Caesar cipher, a fundamental encryption technique, holds historical importance as one of the earliest known methods to protect sensitive information. Named after Julius Caesar, this cipher involves shifting each letter in the plaintext by a fixed number of positions down the alphabet. This simple yet ingenious concept fascinated ancient cryptographers and laid the groundwork for more sophisticated encryption strategies that followed.

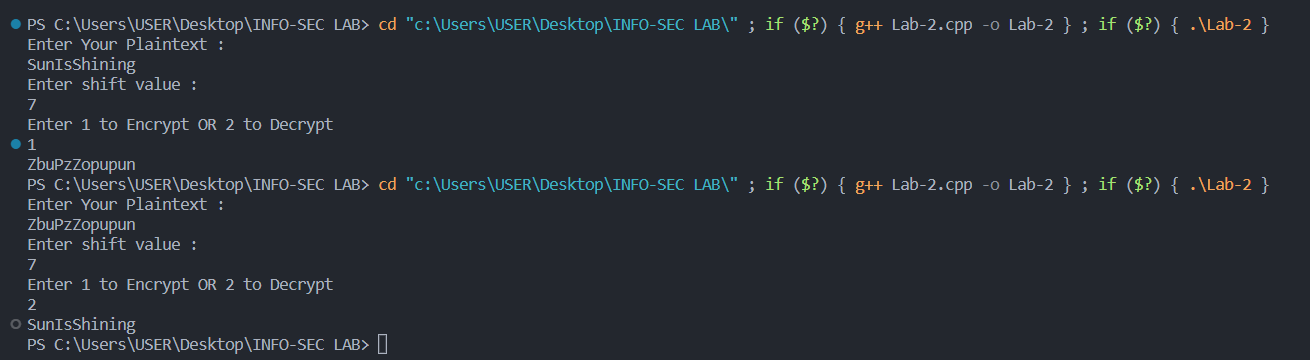
Imagine writing a letter to a friend and then shifting each letter by a certain number of steps. For instance, if you shift the letter 'A' by three steps, it becomes 'D,' and 'B' becomes 'E,' and so on. By applying this shifting pattern to an entire message, the original text transforms into a coded version, known as the ciphertext.

While the Caesar cipher may appear basic, it introduces essential cryptographic concepts. The idea of replacing one letter with another based on a specific rule is at the heart of encryption. The Caesar cipher provides an elementary glimpse into the realm of substitution ciphers, where characters are swapped to create secret messages. This concept serves as a stepping stone for comprehending more intricate encryption techniques that form the foundation of modern information security.

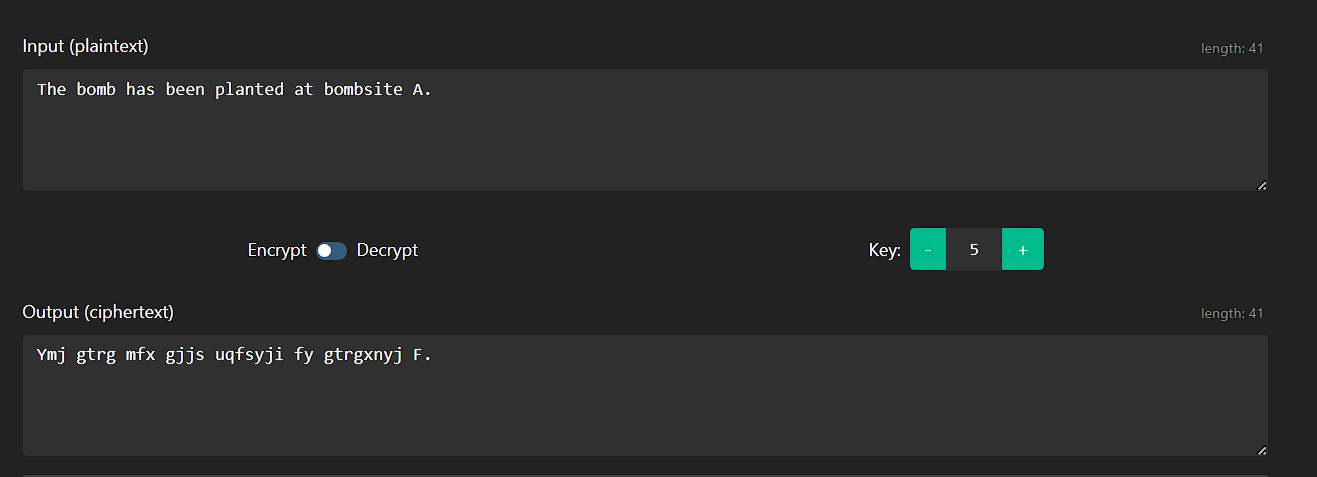
**Program (source code):**

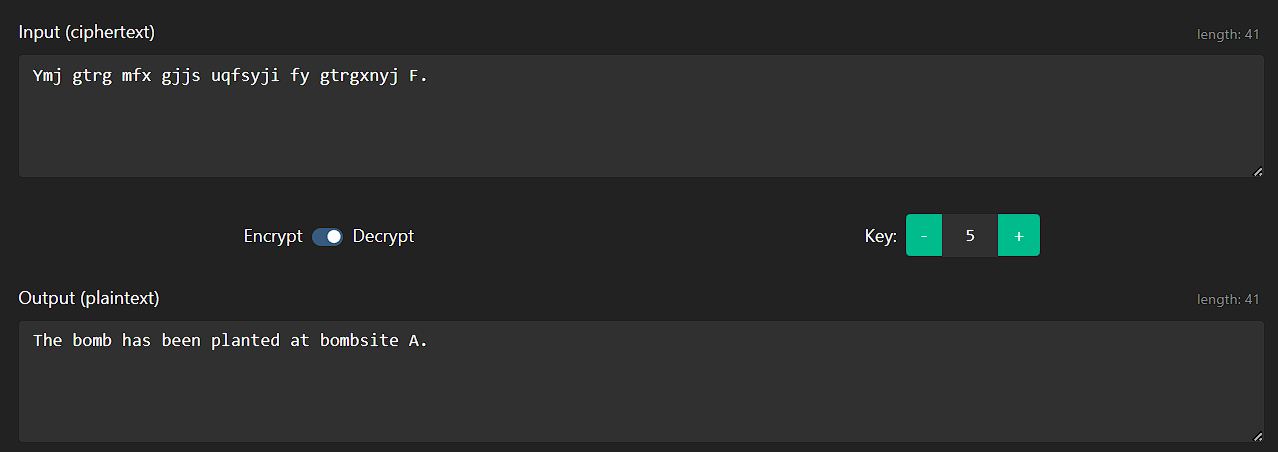
|  |
| --- |
| #include<iostream>  #include<bits/stdc++.h>  using namespace std;  string Encrypt(int cipherKey, string plainText) {      string result = "";      for (int i = 0; i < plainText.length(); i++) {          if (isupper(plainText[i])) {              result += char((plainText[i] + cipherKey - 'A') % 26 + 'A');          } else {              result += char((plainText[i] + cipherKey - 'a') % 26 + 'a');          }      }      return result;  }  string Decrypt(int shift, string plainText){    string result = "";    for(int i = 0; i < plainText.length(); i++){      if(isupper(plainText[i])){        result += char((int(plainText[i] - 'A' - shift + 26) % 26) + 'A');      }      else {        result += char((int(plainText[i] - 'a' - shift + 26) % 26) + 'a');      }    }    return result;  }  int main(){    string plainText;    int shift;    int output;      cout<<"Enter Your Plaintext :"<<endl;    cin>>plainText;    cout<<"Enter shift value : "<<endl;    cin>>shift;    cout<<"Enter 1 to Encrypt OR 2 to Decrypt"<<endl;    cin>>output;    switch (output)    {    case 1:      cout<<Encrypt(shift,plainText)<<endl;;      break;    case 2:      cout<<Decrypt(shift,plainText)<<endl;      break;      default:      cout<<"enter valid operation!!"<<endl;      break;    }    } |

**Output (program):**

****

**Output (Cryptool):**

****

****

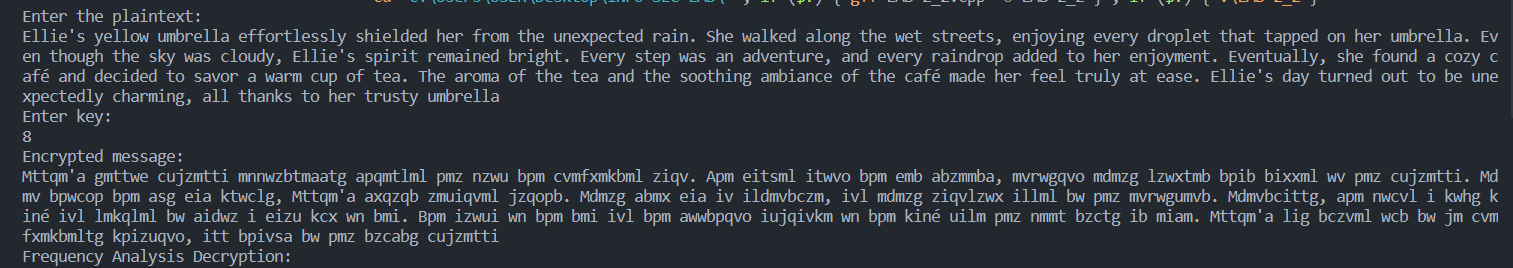
**Cryptanalysis:** Cryptanalysis is the art and science of deciphering encrypted messages without knowledge of the secret key or algorithm. In the context of the Caesar cipher, cryptanalysis involves analyzing the ciphertext to uncover the original plaintext without knowing the shift value used for encryption. While the Caesar cipher was historically effective for its time, modern cryptanalysis techniques have exposed its vulnerabilities.

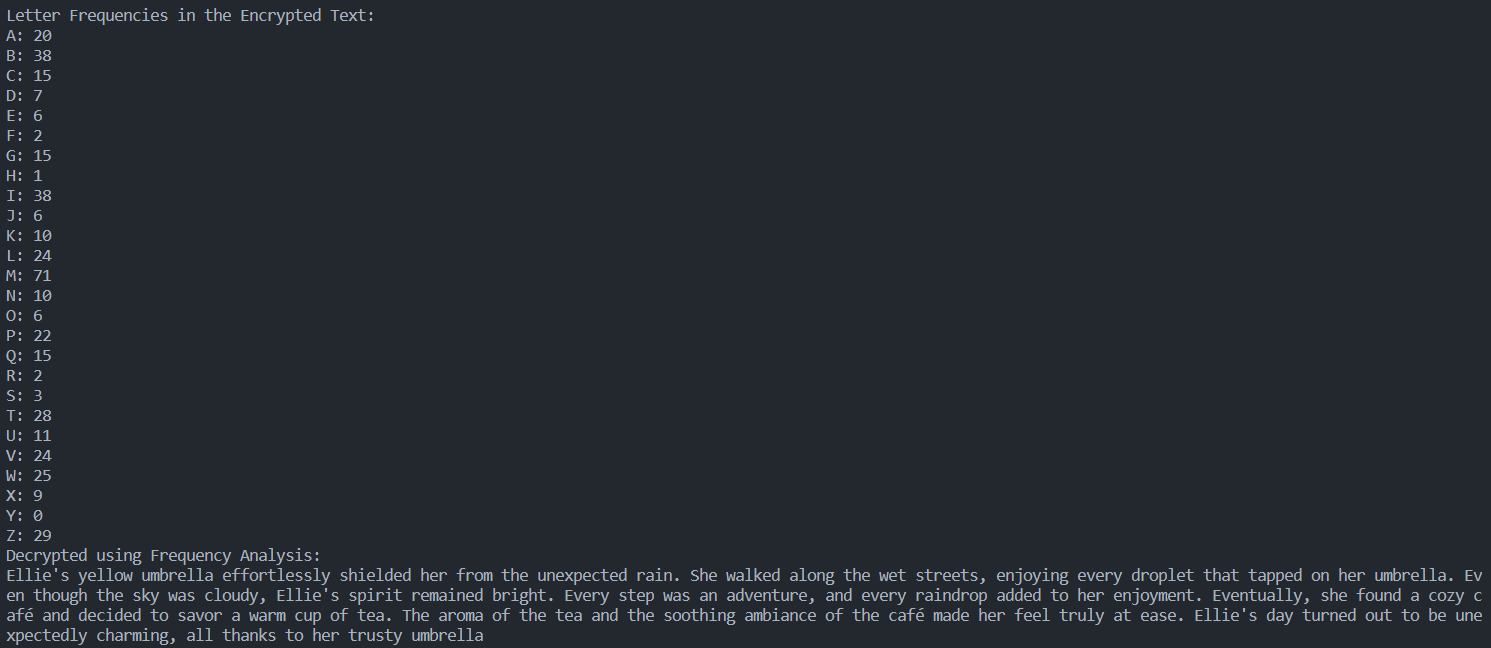
One of the primary weaknesses of the Caesar cipher is its limited key space. With only 25 possible shift values (excluding the no-shift option), an attacker can employ a brute-force attack by trying each possible shift until the correct one is found. The simplicity of the algorithm allows for rapid decryption, making the Caesar cipher insecure for protecting sensitive information.

**Frequency analysis** is another method used to break the Caesar cipher. Since the relative frequencies of letters in English text remain consistent even after encryption, an attacker can analyze the frequency distribution of letters in the ciphertext. By comparing these frequencies to those in the English language, an attacker can deduce the most likely shift value and ultimately decipher the message.

**Example with code & Output:**

|  |
| --- |
| #include <iostream>  #include <string>  #include <algorithm>  using namespace std;  string encrypt(string message, int key) {      string encrypted = "";      for (char ch : message) {          if (isalpha(ch)) {              char base = islower(ch) ? 'a' : 'A';              encrypted += static\_cast<char>((ch - base + key) % 26 + base);          } else {              encrypted += ch;          }      }      return encrypted;  }  string decrypt(string message, int key) {      return encrypt(message, 26 - key);  }  string frequencyAnalysisDecrypt(string message) {      int letterCounts[26] = {0}; // Array to store letter frequencies        for (char c : message) {          if (isalpha(c)) {              char base = islower(c) ? 'a' : 'A';              ++letterCounts[c - base];          }      }        cout << "Letter Frequencies in the Encrypted Text:\n";      for (int i = 0; i < 26; ++i) {          cout << static\_cast<char>('A' + i) << ": " << letterCounts[i] << endl;      }        int mostCommonIndex = distance(letterCounts, max\_element(letterCounts, letterCounts + 26));      int shift = mostCommonIndex - ('E' - 'A');      if (shift < 0) {          shift += 26;      }      return decrypt(message, shift);  }  int main() {      string msg;      int key;      cout << "Enter the plaintext: \n";      getline(cin, msg);      cout << "Enter key: \n";      cin >> key;      cin.ignore();      string encrypted = encrypt(msg, key);      cout << "Encrypted message: \n" << encrypted << "\n";        cout << "Frequency Analysis Decryption:\n";      string frequencyDecryption = frequencyAnalysisDecrypt(encrypted);      cout << "Decrypted using Frequency Analysis: \n" << frequencyDecryption << "\n";      return 0;  } |

****

****

The Caesar cipher's susceptibility to these basic attacks underscores the importance of using more advanced encryption techniques in modern cryptography. Secure encryption algorithms employ larger key spaces, complex mathematical operations, and resistance to cryptanalysis to ensure the confidentiality and integrity of sensitive data.

Understanding the vulnerabilities of the Caesar cipher serves as an educational exercise in cryptanalysis, highlighting the need for stronger encryption methods to safeguard digital communications and data in today's interconnected world.

**Applications:** The Caesar cipher, though outdated and insecure for modern cryptographic requirements, holds historical significance and has found applications in various contexts throughout history. While its limitations prevent it from being used for secure digital communications today, it provides valuable insights into the evolution of cryptography and serves as a teaching tool for understanding fundamental encryption concepts.

1. **Historical Communication Security:** In ancient times, the Caesar cipher played a crucial role in protecting confidential communications. Historical figures, including Julius Caesar, used the cipher to encode military orders, diplomatic messages, and other sensitive information. The cipher's simplicity allowed for efficient encryption and decryption, making it suitable for quick and basic message protection.
2. **Cryptography Education:** The Caesar cipher remains a cornerstone of cryptography education. It serves as an introductory example to demonstrate the fundamental principles of encryption, including substitution and shifting. By understanding the Caesar cipher, students can grasp essential concepts such as plaintext, ciphertext, encryption, and decryption. This foundational knowledge provides a stepping stone for exploring more complex encryption algorithms.
3. **Recreational and Puzzle Applications:** The Caesar cipher continues to be used in recreational and puzzle contexts. Puzzle enthusiasts and educators create challenges and brain teasers involving the Caesar cipher to engage individuals in code-breaking activities. These puzzles encourage critical thinking, logical reasoning, and an appreciation for historical encryption techniques.

**References:**

1. **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>
2. **Wikipedia contributors. (2023)**. *Caesar cipher.* In Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/wiki/Caesar_cipher>
3. **Cryptool.org:** <https://www.cryptool.org/en/cto/caesar>

EXPERIMENT NO – 3

**Aim:** The aim of this document is to provide an overview of the Columnar Transposition Cipher, including its introduction, a sample program's source code for encryption and decryption, program outputs, Cryptool outputs, information on cryptanalysis, its applications, and a list of references for further reading.

**Introduction:** The Columnar Transposition Cipher is a classical cryptographic technique that falls under the category of transposition ciphers. Unlike substitution ciphers that replace individual characters with other characters, the Columnar Transposition Cipher rearranges the characters of the plaintext in a systematic manner to achieve encryption. This rearrangement is done by writing the plaintext into a grid of a specific number of columns and then reading the ciphertext from the grid in a predetermined column order.

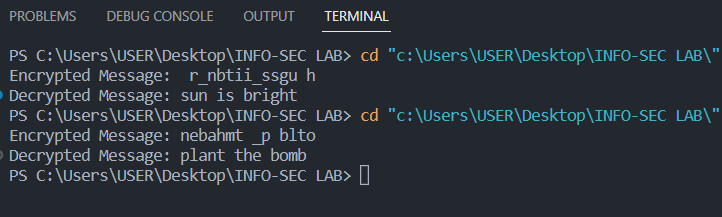
The security of the Columnar Transposition Cipher relies on the complexity of the chosen column order and the key used for encryption. This cipher does not alter the characters themselves; instead, it alters their positions within the grid. While it might not provide the same level of security as modern encryption techniques, it still serves as a valuable tool for understanding basic cryptographic principles and historical encryption methods.

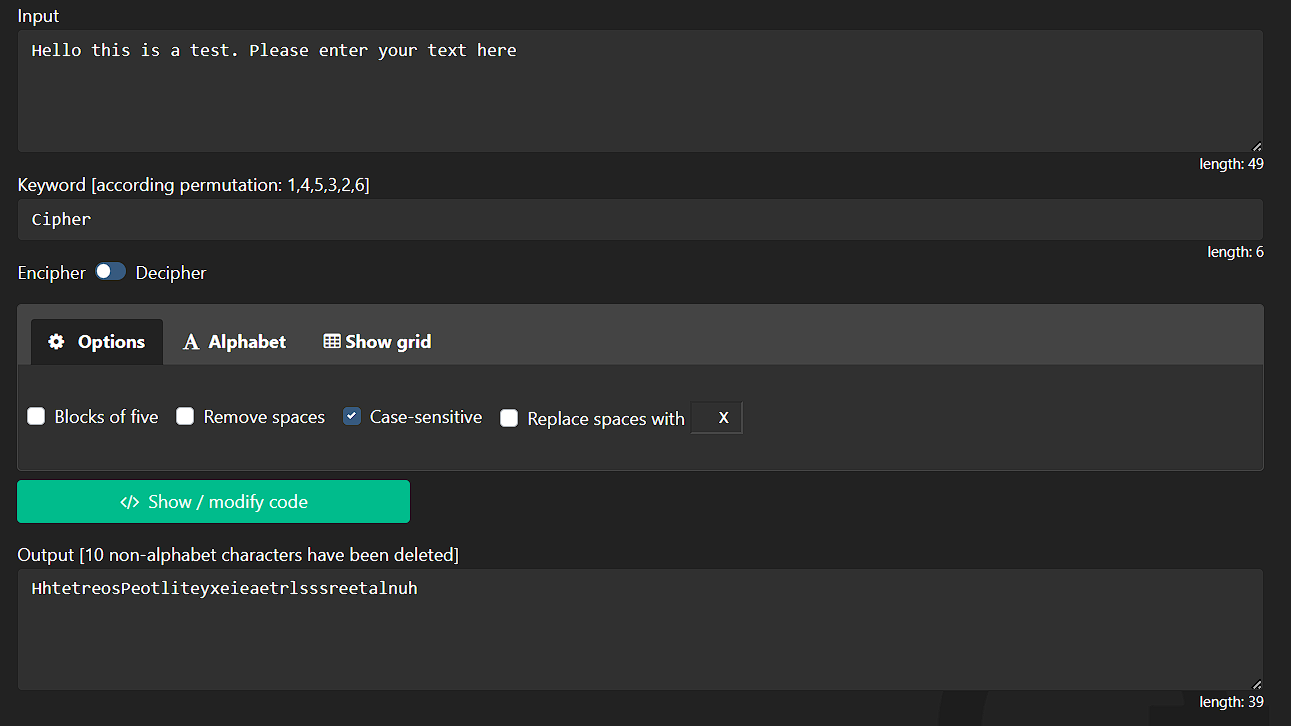
The simplicity of the Columnar Transposition Cipher makes it an excellent educational tool to demonstrate concepts such as encryption, decryption, key management, and the importance of permutation in cryptography. Additionally, this cipher has historical significance, as it was used in various historical periods for secure communication. Despite its vulnerability to modern cryptanalysis techniques, it remains an interesting subject of study in the field of cryptography.

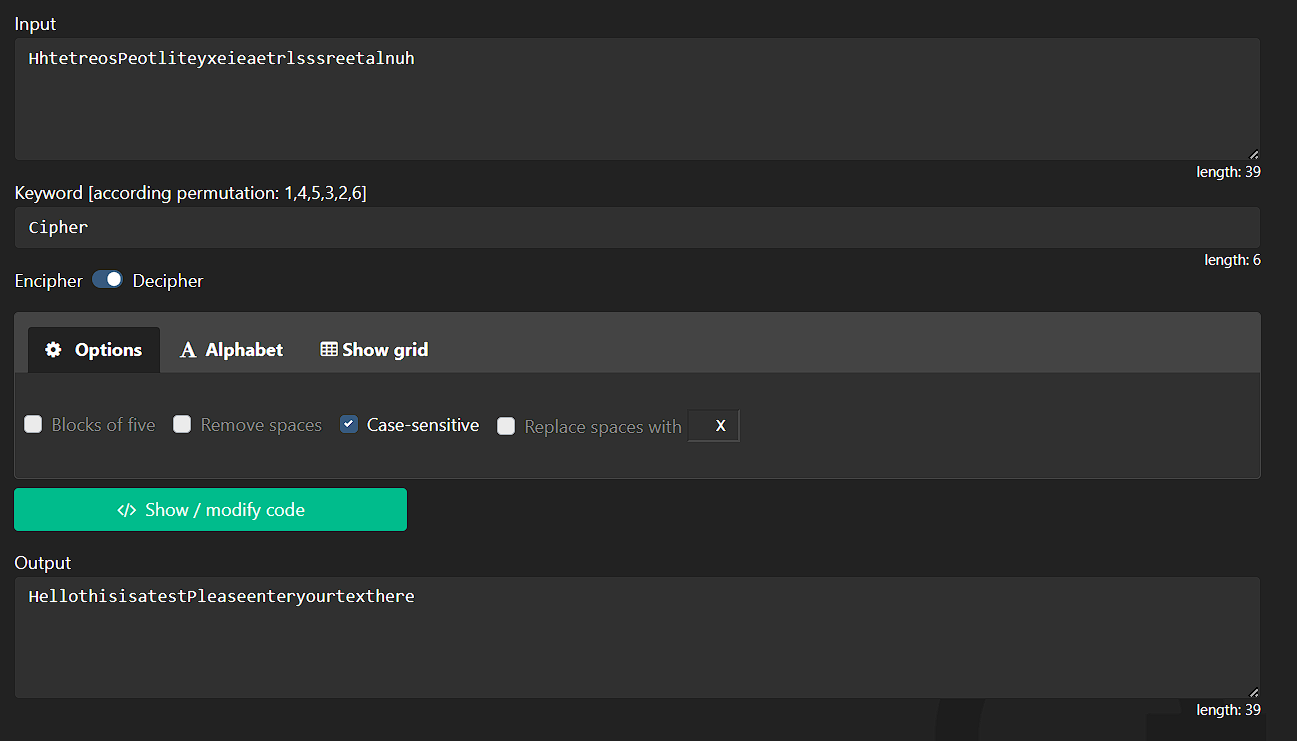
**Program (Source code):**

|  |
| --- |
| #include<bits/stdc++.h>  using namespace std;  string key = "STEAL";  map<int,int> keyTable;  void set\_keyTable(){    for(int i=0;i<key.length();i++){      keyTable[key[i]]= i;    }    }  string encryption(string msg){    int row,col,j;    string cipher\_mat="";    col=key.length(); // total columns    row= msg.length()/col; //total possible rows    if(msg.length() % col){      row=row+1;    }    char matrix[row][col];    for(int i=0,k=0;i<row; i++){      for(int j=0;j<col;){          if(msg[k]=='\0'){            // add the \_ char            matrix[i][j]='\_';            j++;          }          if(isalpha(msg[k]) || msg[k]== ' '){            //add only space and alphabet            matrix[i][j] = msg[k];            j++;          }          k++;      }    }    for(map<int,int> :: iterator i1 = keyTable.begin(); i1!=keyTable.end();++i1){      j=i1->second;        // getting cipher\_mat text from matrix column wise using permuted key      for (int i=0; i<row; i++)      {          if( isalpha(matrix[i][j]) || matrix[i][j]==' ' || matrix[i][j]=='\_')              cipher\_mat += matrix[i][j];      }    }    return cipher\_mat;  }  string decryption(string cipher)  {      // calculate row and column for cipher Matrix      int col = key.length();        int row = cipher.length()/col;      char cipherMat[row][col];        // add character into matrix column wise      for (int j=0,k=0; j<col; j++)          for (int i=0; i<row; i++)              cipherMat[i][j] = cipher[k++];        // update the order of key for decryption      int index = 0;      for( map<int,int>::iterator ii=keyTable.begin(); ii!=keyTable.end(); ++ii)          ii->second = index++;        // Arrange the matrix column wise according to permutation order by adding into new matrix        char decCipher[row][col];      map<int,int>::iterator ii=keyTable.begin();      int k = 0;      for (int l=0,j; key[l]!='\0'; k++)      {          j = keyTable[key[l++]];          for (int i=0; i<row; i++)          {              decCipher[i][k]=cipherMat[i][j];          }      }        // getting Message using matrix      string msg = "";      for (int i=0; i<row; i++)      {          for(int j=0; j<col; j++)          {              if(decCipher[i][j] != '\_')                  msg += decCipher[i][j];          }      }      return msg;  }    int main(){    string msg = "plant the bomb";        set\_keyTable();        string cipher = encryption(msg);      cout << "Encrypted Message: " << cipher << endl;        cout << "Decrypted Message: " << decryption(cipher) << endl;        return 0;  } |

**Output (Program):**



**Output (Cryptool):**

****

**Cryptanalysis:** Cryptanalysis of the Columnar Transposition Cipher involves deciphering an encrypted message without the knowledge of the key or the column order used for encryption. While this cipher was effective in historical contexts, it can be vulnerable to various cryptanalytic techniques, particularly due to its limited complexity compared to modern encryption methods.

1. **Brute Force Attack:** Since the Columnar Transposition Cipher has a finite number of possible key permutations and column orders, a brute force attack involves trying out all possible combinations until the correct one is found. However, this approach becomes time-consuming as the length of the key and the message increases.

2. **Frequency Analysis:** Although the Columnar Transposition Cipher does not alter the characters themselves, it can still reveal some patterns in the ciphertext. For instance, if the plaintext message has repeating patterns or frequent letter combinations, these patterns might become visible in the ciphertext, aiding frequency analysis.

3. **Known-Plaintext Attack:** If an attacker has access to both the plaintext and the corresponding ciphertext (known-plaintext attack), they can attempt to deduce the column order and key used for encryption. This method becomes more effective if the message contains repeated words or phrases.

4. **Chosen-Plaintext Attack:** In a chosen-plaintext attack scenario, the attacker can choose specific plaintext inputs and observe their corresponding ciphertext outputs. By analyzing the resulting ciphertext patterns, an attacker might gain insights into the key and column arrangement.

5. **Exploiting Key Lengths:** If the length of the key used for the Columnar Transposition Cipher is short, the number of possible keys and column orders is limited. This vulnerability can be exploited by an attacker to perform an exhaustive search over all possible keys.

6. **Cryptanalysis Tools:** Modern cryptanalysis tools and software can aid in decrypting Columnar Transposition Cipher-encrypted messages. Algorithms that test various key permutations and analyze the resulting ciphertext patterns can significantly speed up the process of breaking the cipher.

It's important to note that the security of the Columnar Transposition Cipher depends on the key length, the choice of column order, and the complexity of the plaintext. While the cipher might offer a certain level of security against casual attempts to decipher the message, it is not considered secure against determined attackers armed with modern cryptanalytic techniques.

**Applications:**

Despite its vulnerabilities to modern cryptanalysis techniques, the Columnar Transposition Cipher has been used historically and continues to find applications in various contexts due to its simplicity and educational value:

1. **Historical Usage:** The Columnar Transposition Cipher was employed in historical periods when encryption methods were limited. It was used to secure military communications and sensitive information during times when more advanced encryption techniques were not available.

2. **Educational Tool:** The simplicity of the Columnar Transposition Cipher makes it an excellent tool for introducing students and enthusiasts to fundamental concepts of cryptography. It helps them understand the principles of permutation, transposition, encryption, and decryption.

3. **Puzzles and Challenges:** The Columnar Transposition Cipher is sometimes used in puzzles, cryptography challenges, and escape room scenarios. Participants are tasked with deciphering messages encrypted using this cipher, adding an element of intrigue and problem-solving to the experience.

4. **Basic Encryption Demonstrations:** In educational settings, the Columnar Transposition Cipher can be used to demonstrate how encryption and decryption work. Its straightforward implementation helps learners grasp the concept of reordering characters to achieve confidentiality.

5. **Algorithmic Understanding:** Computer science and mathematics students can use the Columnar Transposition Cipher as a practical example to explore permutation algorithms, indexing techniques, and the principles of designing cryptographic algorithms.

6. **Creative Writing and Literature:** Writers and authors interested in embedding hidden messages or codes within their works might use the Columnar Transposition Cipher as a creative way to engage readers and create puzzles.

While the Columnar Transposition Cipher may not offer strong security in modern cryptographic contexts, its applications remain valuable in educational, historical, and recreational settings. It serves as a reminder of the evolution of encryption techniques and the importance of adapting to ever-advancing methods of encryption and cryptanalysis.

**References:**

1. **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>
2. **Wikipedia contributors. (2023)** Transposition cipher*.* In Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/wiki/Transposition_cipher>
3. **Cryptool.org:** <https://www.cryptool.org/en/cto/transposition>

EXPERIMENT NO – 4

**Aim:** The aim of this experiment is to understand the Rail Fence Cipher encryption and decryption process and to gain hands-on experience in implementing and analyzing this classic transposition cipher.

**Introduction:** The Rail Fence Cipher is a simple transposition cipher that rearranges the letters of a plaintext message to create a ciphertext. It is named after its visual representation, where the letters are arranged diagonally in a zigzag pattern resembling a fence made of rails. This cipher is easy to understand and implement, making it suitable for educational purposes.

In this cipher, the plaintext is written in a zigzag pattern along a set number of "rails" or rows. Then, the ciphertext is created by reading the letters in a row-wise manner. The number of rails and the order in which they are filled determine the encryption key. Decryption is performed by reversing this process.

// Rail Fence Cipher

#include <bits/stdc++.h>

using namespace std;

// function to encrypt a message

string **encryptRailFence**(string text, int key)

{

  // create the matrix

  char rail[key][(text.**length**())];

  // fill the rail matrix to distinguish filled

  // spaces from blank ones

  for (int i=0; i < key; i++)

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

      rail[i][j] = '\n';

  // to find the direction

  bool dir\_down = false;

  int row = 0, col = 0;

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

  {

    // check the direction of flow

    // reverse the direction if we've just

    // filled the top or bottom rail

    if (row == 0 || row == key-1)

      dir\_down = !dir\_down;

    // fill the corresponding alphabet

    rail[row][col++] = text**[**i**]**;

    // find the next row using direction flag

    dir\_down?row++ : row--;

  }

  // construct the cipher using the rail matrix

  string result;

  for (int i=0; i < key; i++)

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

      if (rail[i][j]!='\n')

        result.**push\_back**(rail[i][j]);

  return result;

}

// This function receives cipher-text and key

// and returns the original text after decryption

string **decryptRailFence**(string cipher, int key)

{

  // create the matrix to cipher plain text

  // key = rows , length(text) = columns

  char rail[key][cipher.**length**()];

  // filling the rail matrix to distinguish filled

  // spaces from blank ones

  for (int i=0; i < key; i++)

    for (int j=0; j < cipher.**length**(); j++)

      rail[i][j] = '\n';

  // to find the direction

  bool dir\_down;

  int row = 0, col = 0;

  // mark the places with '\*'

  for (int i=0; i < cipher.**length**(); i++)

  {

    // check the direction of flow

    if (row == 0)

      dir\_down = true;

    if (row == key-1)

      dir\_down = false;

    // place the marker

    rail[row][col++] = '\*';

    // find the next row using direction flag

    dir\_down?row++ : row--;

  }

  // now we can construct the fill the rail matrix

  int index = 0;

  for (int i=0; i<key; i++)

    for (int j=0; j<cipher.**length**(); j++)

      if (rail[i][j] == '\*' && index<cipher.**length**())

        rail[i][j] = cipher**[**index++**]**;

  // now read the matrix in zig-zag manner to construct

  // the resultant text

  string result;

  row = 0, col = 0;

  for (int i=0; i< cipher.**length**(); i++)

  {

    // check the direction of flow

    if (row == 0)

      dir\_down = true;

    if (row == key-1)

      dir\_down = false;

    // place the marker

    if (rail[row][col] != '\*')

      result.**push\_back**(rail[row][col++]);

    // find the next row using direction flag

    dir\_down?row++: row--;

  }

  return result;

}

//driver program to check the above functions

int **main**()

{

  cout **<<** **encryptRailFence**("attack  postpone", 2) **<<** **endl**;

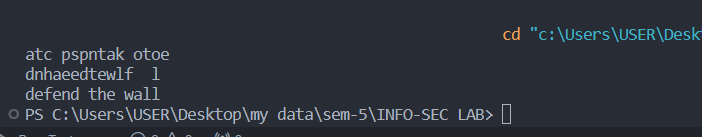
  cout **<<** **encryptRailFence**("defend the wall", 3) **<<** **endl**;

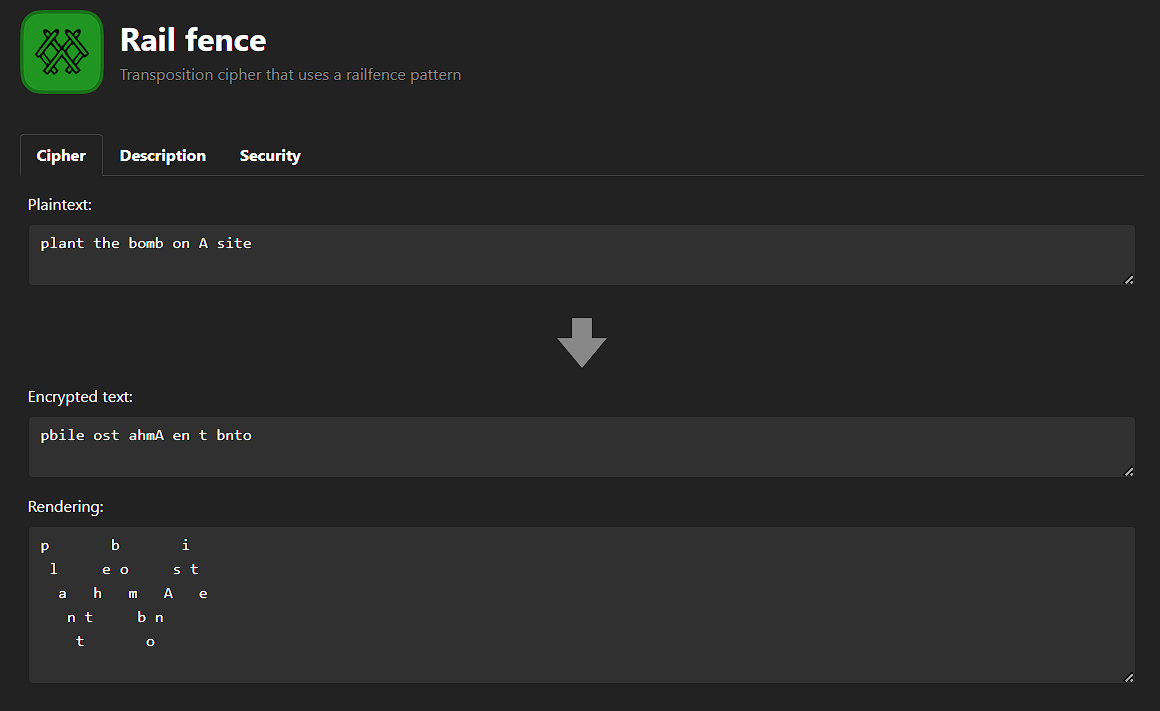
  //Now decryption of the same cipher-text

  cout **<<** **decryptRailFence**("dnhaeedtewlf  l",3) **<<** **endl**;

  return 0;

}





**Cryptanalysis:**  
The Rail Fence Cipher, also known as the Zigzag Cipher, is a basic transposition cipher that rearranges the characters in a message to make it harder to read. It's called "Rail Fence" because when you write out the letters in a zigzag pattern, they resemble the rails of a fence. While it can be a fun and interesting way to encode messages, it is indeed relatively weak in terms of security. Let's delve into the reasons why this cipher is vulnerable to cryptanalysis.

1. **Lack of Strong Encryption:** The Rail Fence Cipher relies solely on transposing the letters of the plaintext without any substitution of characters or complex mathematical operations. This means that the original letters remain the same, only their positions change. As a result, it doesn't provide strong encryption against modern cryptanalysis techniques.
2. **Vulnerability to Frequency Analysis**: Frequency analysis is a common technique used to break simple ciphers like the Rail Fence Cipher. In frequency analysis, an attacker examines the frequency of letters or letter pairs in the ciphertext. Because the Rail Fence Cipher doesn't change the letters but merely rearranges them, the frequency distribution of letters in the ciphertext still roughly matches that of the language being used (e.g., English). This allows attackers to make educated guesses about the original text.
3. **Dependency on the Number of Rails:** One of the most significant weaknesses of the Rail Fence Cipher is that the security depends on the number of rails used. When you encrypt a message, you specify the number of rails (rows) to use in the zigzag pattern. If the attacker knows the number of rails, it becomes much easier to decrypt the message. They can simply try different rail counts until they find one that produces meaningful text. This process is essentially a brute-force attack and is feasible because there are only a limited number of possible rail counts.

Here's a step-by-step description of how an attacker might break the Rail Fence Cipher:

1. Determine the Number of Rails: If the attacker knows the number of rails used, they have a significant advantage. If not, they may try different rail counts to see which one produces a readable result.
2. Perform Frequency Analysis: The attacker analyzes the frequency of letters or letter pairs in the ciphertext. They look for patterns that resemble common letter frequencies in the language of the plaintext (e.g., 'E' is the most common letter in English).
3. Trial and Error: Using the rail count and frequency analysis, the attacker tries different rail configurations until they find one that produces meaningful words or phrases. They can manually inspect the results to determine if they have successfully decrypted the message.

**Applications:** The Rail Fence Cipher is not suitable for secure communication, but it can be used for educational purposes to introduce the concept of transposition ciphers and basic encryption techniques. It may also be used in puzzle games or as a simple encoding method for fun activities. In historical contexts, it has been used for non-critical, low-security communications.

**References:**

1. **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>
2. **Wikipedia contributors. (2023)** In Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/wiki/Rail_fence_cipher>

**Cryptool.org:** [https://www.cryptool.org/en/](https://www.cryptool.org/en/cto/transposition)

EXPERIMENT NO – 5

**Aim:** The primary aim of this experiment is to explore and understand the Vigenère Cipher, a polyalphabetic substitution cipher, through practical implementation. This experiment aims to teach students the principles of polyalphabetic ciphers and their application in encryption and decryption.

**Introduction:** The Vigenère Cipher is a classical polyalphabetic substitution cipher that enhances the security of simple substitution ciphers like the Caesar Cipher. In this cipher, instead of applying a single fixed shift to all characters in the plaintext, a keyword is used to determine the shifting for each character. This added layer of complexity makes the Vigenère Cipher more resistant to frequency analysis and other cryptanalytic techniques.

The encryption process in the Vigenère Cipher involves repeating the keyword as needed to match the length of the plaintext. Each character in the plaintext is then shifted by the corresponding character in the keyword. Decryption is performed by reversing these shifts using the same keyword.

// C++ code to implement Vigenere Cipher

#include<bits/stdc++.h>

using namespace std;

// This function generates the key in

// a cyclic manner until it's length isn't

// equal to the length of original text

string **generateKey**(string str, string key)

{

  int x = str.**size**();

  for (int i = 0; ; i++)

  {

    if (x == i)

      i = 0;

    if (key.**size**() == str.**size**())

      break;

    key.**push\_back**(key[i]);

  }

  return key;

}

// This function returns the encrypted text

// generated with the help of the key

string **cipherText**(string str, string key)

{

  string cipher\_text;

  for (int i = 0; i < str.**size**(); i++)

  {

    // converting in range 0-25

    char x = (str[i] + key[i]) %26;

    // convert into alphabets(ASCII)

    x += 'A';

    cipher\_text.**push\_back**(x);

  }

  return cipher\_text;

}

// This function decrypts the encrypted text

// and returns the original text

string **originalText**(string cipher\_text, string key)

{

  string orig\_text;

  for (int i = 0 ; i < cipher\_text.**size**(); i++)

  {

    // converting in range 0-25

    char x = (cipher\_text[i] - key[i] + 26) %26;

    // convert into alphabets(ASCII)

    x += 'A';

    orig\_text.**push\_back**(x);

  }

  return orig\_text;

}

// Driver program to test the above function

int **main**()

{

  string str = "KARTIKPATEL";

  string keyword = "HEHE";

  string key = **generateKey**(str, keyword);

  string cipher\_text = **cipherText**(str, key);

  cout << "Ciphertext : "

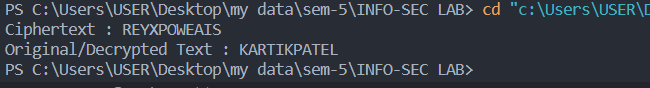
    << cipher\_text << "\n";

  cout << "Original/Decrypted Text : "

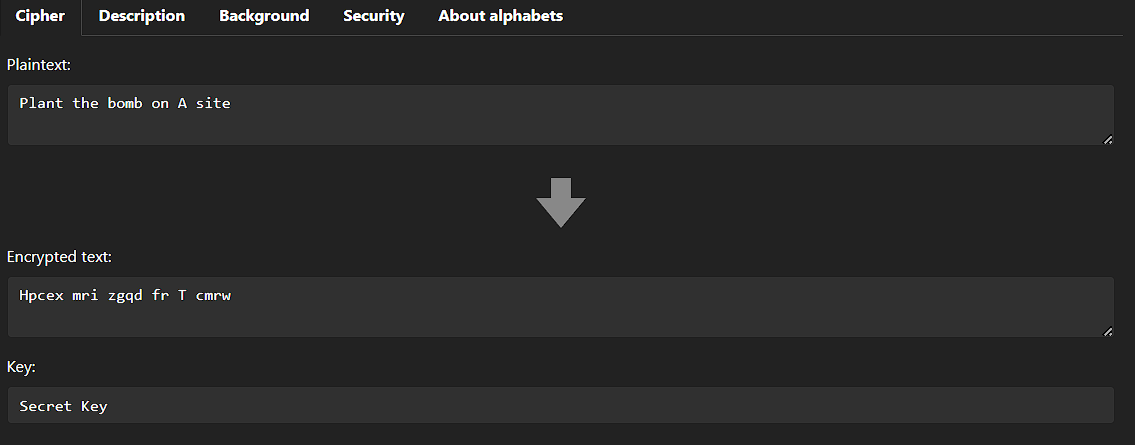
    << **originalText**(cipher\_text, key);

  return 0;

}



**Cryptool:**



The Vigenère Cipher is a polyalphabetic substitution cipher that adds complexity to the traditional Caesar Cipher by using a keyword to determine the shift for each letter in the plaintext. While it provides more security compared to simple substitution ciphers, it is still vulnerable to various cryptanalysis techniques. Below, I'll provide a detailed cryptanalysis of the Vigenère Cipher.

1. Kasiski Examination:
   * This technique involves looking for repeating patterns in the ciphertext. If the same sequence of letters appears in the ciphertext at regular intervals, it might indicate that the same part of the keyword is used to encrypt those letters.
   * To perform Kasiski examination, you can:
     + Identify repeated sequences in the ciphertext.
     + Calculate the distances between these repetitions.
     + Look for common factors among the distances, which can be indicative of the keyword's length.
2. Frequency Analysis:
   * While Vigenère Cipher makes frequency analysis more challenging compared to simple substitution ciphers, it's not immune to it. Since different parts of the ciphertext are encrypted using different Caesar ciphers (based on the keyword), you can perform frequency analysis on the letters of the ciphertext modulo the keyword length.
   * For example, if you suspect that a 5-letter keyword is used, you can split the ciphertext into five groups, each containing letters that were encrypted using the same Caesar cipher. Then, you can analyze the frequency of letters in each group.
3. Guessing the Keyword Length:
   * Once you've identified the potential keyword length using Kasiski examination or other methods, you can try to guess the actual keyword. This often involves trying common English words or phrases as potential keywords.
   * If you've correctly guessed the keyword length and keyword, you can decrypt a portion of the message and use context to guess other parts of the keyword.
4. Known-Plaintext Attack:
   * If you have access to some portions of the plaintext and their corresponding ciphertext, you can use these pairs to deduce parts of the keyword and then extend your knowledge of the keyword.
   * For example, if you know that a certain word appears at a specific position in both the plaintext and ciphertext, you can determine the keyword characters that produced those letters.
5. Brute Force Attack:
   * If all else fails, you can perform a brute force attack by trying all possible keyword combinations. The number of combinations increases exponentially with the length of the keyword, so this approach becomes less feasible for longer keywords.
   * However, if the keyword is very short, this method can be effective.
6. Dictionary Attack:
   * If the keyword is a common word or phrase, you can try using dictionary-based attacks. In this approach, you systematically try known words or phrases as the keyword.
   * This approach is often used when the attacker suspects that the keyword is a word from the English language.

Using a longer keyword can significantly enhance the security of the Vigenère Cipher.

**Applications:** The Vigenère Cipher has historical significance as one of the earliest methods of encryption. In modern times, it is mainly used for educational purposes and to illustrate the concept of polyalphabetic substitution ciphers. It is not suitable for secure communication in today's cryptographic landscape due to its vulnerability to cryptanalysis. Instead, it serves as a foundation for understanding more advanced encryption techniques.

**References:**

1. **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>
2. **Wikipedia contributors. (2023)***.* In Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher>

**Cryptool.org:** [https://www.cryptool.org/en/](https://www.cryptool.org/en/cto/transposition)

EXPERIMENT NO – 6

**Aim:** The primary aim of this lab experiment is to introduce students to the Playfair Cipher, a classical digraphic substitution cipher, and provide hands-on experience in encryption and decryption using this historical cryptographic technique.

**Introduction:** The Playfair Cipher, invented by Charles Wheatstone in 1854 but popularized by Lyon Playfair, is a digraphic substitution cipher that enhances the security of simple monoalphabetic ciphers. It operates on pairs of letters (digraphs) instead of individual letters, adding complexity to the encryption process.

The key to the Playfair Cipher is a keyword or key phrase that is used to create a 5x5 matrix, known as the Playfair Square or Playfair Key Table. The matrix is filled with unique letters from the key, omitting any duplicates. The remaining letters of the alphabet are then filled in, excluding any letters from the key already in the matrix. The encryption and decryption processes involve replacing pairs of letters in the plaintext with corresponding pairs from the Playfair Square.

**Implementation:**

// Playfair Cipher

#include <bits/stdc++.h>

using namespace std;

#define **SIZE** 30

// Function to convert the string to lowercase

void **toLowerCase**(char plain[], int ps)

{

  int i;

  for (i = 0; i < ps; i++) {

    if (plain[i] > 64 && plain[i] < 91)

      plain[i] += 32;

  }

}

// Function to remove all spaces in a string

int **removeSpaces**(char\* plain, int ps)

{

  int i, count = 0;

  for (i = 0; i < ps; i++)

    if (plain[i] != ' ')

      plain[count++] = plain[i];

  plain[count] = '\0';

  return count;

}

// Function to generate the 5x5 key square

void **generateKeyTable**(char key[], int ks, char keyT[5][5])

{

  int i, j, k, flag = 0;

  // a 26 character hashmap

  // to store count of the alphabet

  int dicty[26] = { 0 };

  for (i = 0; i < ks; i++) {

    if (key[i] != 'j')

      dicty[key[i] - 97] = 2;

  }

  dicty['j' - 97] = 1;

  i = 0;

  j = 0;

  for (k = 0; k < ks; k++) {

    if (dicty[key[k] - 97] == 2) {

      dicty[key[k] - 97] -= 1;

      keyT[i][j] = key[k];

      j++;

      if (j == 5) {

        i++;

        j = 0;

      }

    }

  }

  for (k = 0; k < 26; k++) {

    if (dicty[k] == 0) {

      keyT[i][j] = (char)(k + 97);

      j++;

      if (j == 5) {

        i++;

        j = 0;

      }

    }

  }

}

// Function to search for the characters of a digraph

// in the key square and return their position

void **search**(char keyT[5][5], char a, char b, int arr[])

{

  int i, j;

  if (a == 'j')

    a = 'i';

  else if (b == 'j')

    b = 'i';

  for (i = 0; i < 5; i++) {

    for (j = 0; j < 5; j++) {

      if (keyT[i][j] == a) {

        arr[0] = i;

        arr[1] = j;

      }

      else if (keyT[i][j] == b) {

        arr[2] = i;

        arr[3] = j;

      }

    }

  }

}

// Function to find the modulus with 5

int **mod5**(int a) { return (a % 5); }

// Function to make the plain text length to be even

int **prepare**(char str[], int ptrs)

{

  if (ptrs % 2 != 0) {

    str[ptrs++] = 'z';

    str[ptrs] = '\0';

  }

  return ptrs;

}

// Function for performing the encryption

void **encrypt**(char str[], char keyT[5][5], int ps)

{

  int i, a[4];

  for (i = 0; i < ps; i += 2) {

**search**(keyT, str[i], str[i + 1], a);

    if (a[0] == a[2]) {

      str[i] = keyT[a[0]][**mod5**(a[1] + 1)];

      str[i + 1] = keyT[a[0]][**mod5**(a[3] + 1)];

    }

    else if (a[1] == a[3]) {

      str[i] = keyT[**mod5**(a[0] + 1)][a[1]];

      str[i + 1] = keyT[**mod5**(a[2] + 1)][a[1]];

    }

    else {

      str[i] = keyT[a[0]][a[3]];

      str[i + 1] = keyT[a[2]][a[1]];

    }

  }

}

// Function to encrypt using Playfair Cipher

void **encryptByPlayfairCipher**(char str[], char key[])

{

  char ps, ks, keyT[5][5];

  // Key

  ks = **strlen**(key);

  ks = **removeSpaces**(key, ks);

**toLowerCase**(key, ks);

  // Plaintext

  ps = **strlen**(str);

**toLowerCase**(str, ps);

  ps = **removeSpaces**(str, ps);

  ps = **prepare**(str, ps);

**generateKeyTable**(key, ks, keyT);

**encrypt**(str, keyT, ps);

}

// Driver code

int **main**()

{

  char str[**SIZE**], key[**SIZE**];

  // Key to be encrypted

**strcpy**(key, "attacknow");

  cout **<<** "Key text: " **<<** key **<<** "\n";

  // Plaintext to be encrypted

**strcpy**(str, "rightnow");

  cout **<<** "Plain text: " **<<** str **<<** "\n";

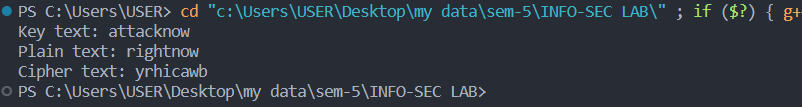
  // encrypt using Playfair Cipher

**encryptByPlayfairCipher**(str, key);

  cout **<<** "Cipher text: " **<<** str **<<** "\n";

  return 0;

}



#include <bits/stdc++.h>

using namespace std;

#define **SIZE** 30

// Convert all the characters

// of a string to lowercase

void **toLowerCase**(char plain[], int ps)

{

  int i;

  for (i = 0; i < ps; i++) {

    if (plain[i] > 64 && plain[i] < 91)

      plain[i] += 32;

  }

}

// Remove all spaces in a string

// can be extended to remove punctuation

int **removeSpaces**(char\* plain, int ps)

{

  int i, count = 0;

  for (i = 0; i < ps; i++)

    if (plain[i] != ' ')

      plain[count++] = plain[i];

  plain[count] = '\0';

  return count;

}

// generates the 5x5 key square

void **generateKeyTable**(char key[], int ks, char keyT[5][5])

{

  int i, j, k, flag = 0, \*dicty;

  // a 26 character hashmap

  // to store count of the alphabet

  dicty = (int\*)**calloc**(26, sizeof(int));

  for (i = 0; i < ks; i++) {

    if (key[i] != 'j')

      dicty[key[i] - 97] = 2;

  }

  dicty['j' - 97] = 1;

  i = 0;

  j = 0;

  for (k = 0; k < ks; k++) {

    if (dicty[key[k] - 97] == 2) {

      dicty[key[k] - 97] -= 1;

      keyT[i][j] = key[k];

      j++;

      if (j == 5) {

        i++;

        j = 0;

      }

    }

  }

  for (k = 0; k < 26; k++) {

    if (dicty[k] == 0) {

      keyT[i][j] = (char)(k + 97);

      j++;

      if (j == 5) {

        i++;

        j = 0;

      }

    }

  }

}

// Search for the characters of a digraph

// in the key square and return their position

void **search**(char keyT[5][5], char a, char b, int arr[])

{

  int i, j;

  if (a == 'j')

    a = 'i';

  else if (b == 'j')

    b = 'i';

  for (i = 0; i < 5; i++) {

    for (j = 0; j < 5; j++) {

      if (keyT[i][j] == a) {

        arr[0] = i;

        arr[1] = j;

      }

      else if (keyT[i][j] == b) {

        arr[2] = i;

        arr[3] = j;

      }

    }

  }

}

// Function to find the modulus with 5

int **mod5**(int a)

{

  if (a < 0)

    a += 5;

  return (a % 5);

}

// Function to decrypt

void **decrypt**(char str[], char keyT[5][5], int ps)

{

  int i, a[4];

  for (i = 0; i < ps; i += 2) {

**search**(keyT, str[i], str[i + 1], a);

    if (a[0] == a[2]) {

      str[i] = keyT[a[0]][**mod5**(a[1] - 1)];

      str[i + 1] = keyT[a[0]][**mod5**(a[3] - 1)];

    }

    else if (a[1] == a[3]) {

      str[i] = keyT[**mod5**(a[0] - 1)][a[1]];

      str[i + 1] = keyT[**mod5**(a[2] - 1)][a[1]];

    }

    else {

      str[i] = keyT[a[0]][a[3]];

      str[i + 1] = keyT[a[2]][a[1]];

    }

  }

}

// Function to call decrypt

void **decryptByPlayfairCipher**(char str[], char key[])

{

  char ps, ks, keyT[5][5];

  // Key

  ks = **strlen**(key);

  ks = **removeSpaces**(key, ks);

**toLowerCase**(key, ks);

  // ciphertext

  ps = **strlen**(str);

**toLowerCase**(str, ps);

  ps = **removeSpaces**(str, ps);

**generateKeyTable**(key, ks, keyT);

**decrypt**(str, keyT, ps);

}

// Driver code

int **main**()

{

  char str[**SIZE**], key[**SIZE**];

  // Key to be encrypted

**strcpy**(key, "ATTACKPOSTPONE");

  cout **<<** "Key Text: " **<<** key **<<** **endl**;

  // Ciphertext to be decrypted

**strcpy**(str, "dcamuwaqqmcw");

  cout **<<** "Plain text: " **<<** str **<<** **endl**;

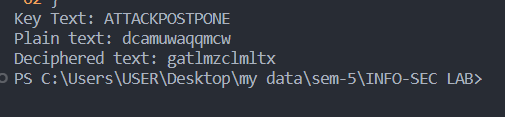
  // encrypt using Playfair Cipher

**decryptByPlayfairCipher**(str, key);

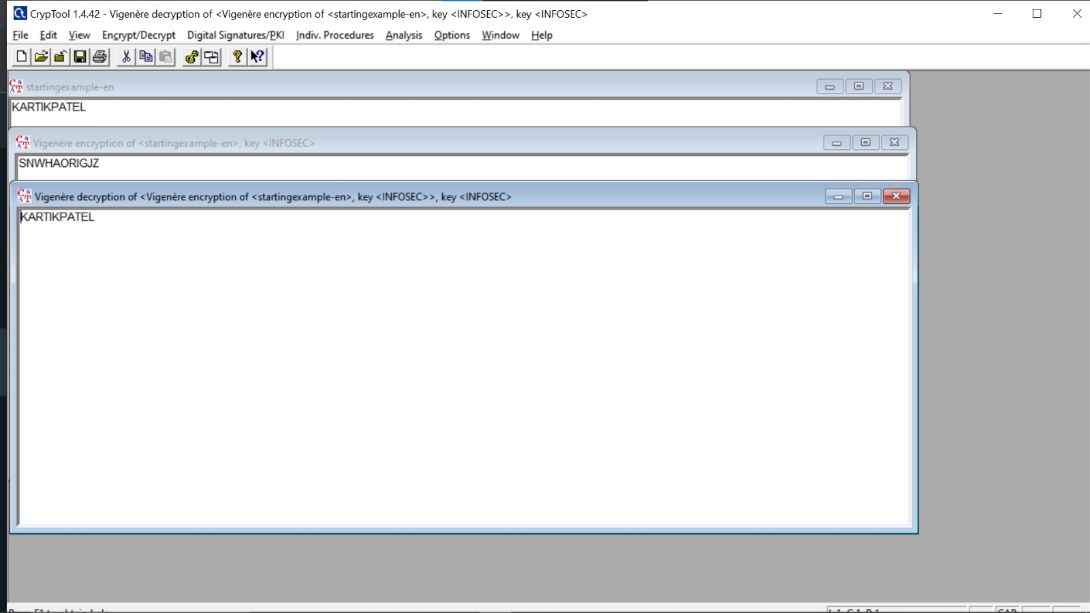
  cout **<<** "Deciphered text: " **<<** str **<<** **endl**;

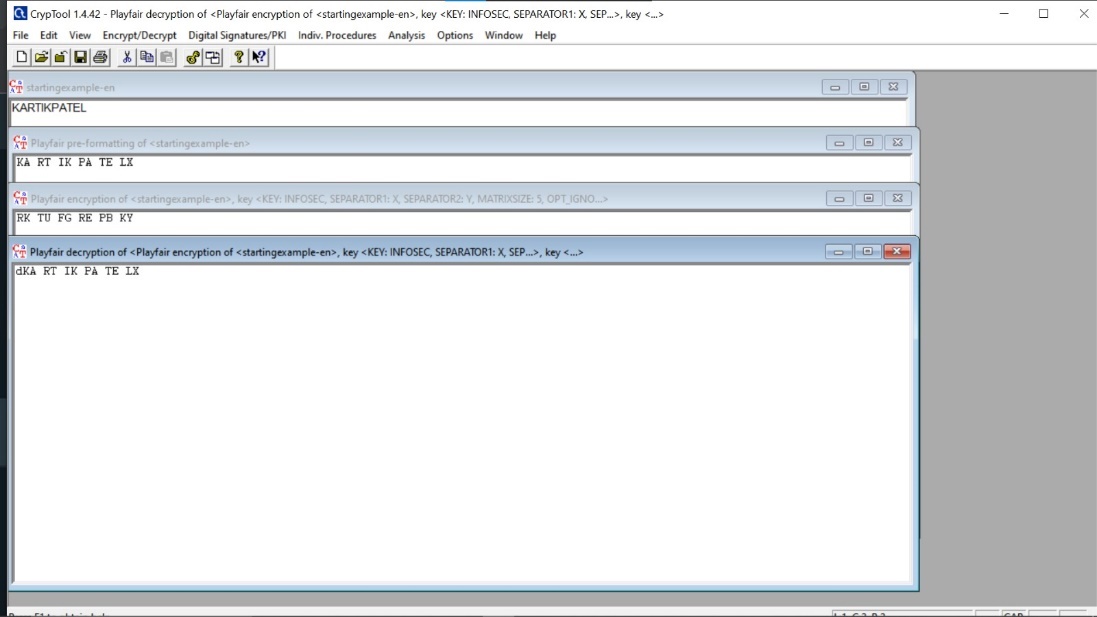
  return 0;

}



**Cryptool:**





**Cryptanalysis:** The Playfair Cipher provides stronger security than simple substitution ciphers because it operates on digraphs, making frequency analysis more challenging. However, it is not immune to cryptanalysis. Some common techniques for breaking Playfair-encrypted messages include:

1. **Frequency Analysis:** While Playfair reduces the frequency of individual letters, it does not eliminate frequency patterns in digraphs. Frequency analysis can still be applied to identify common digraphs.
2. **Brute Force:** Given the limited number of possible keys, a brute-force attack is feasible for smaller Playfair Square sizes.
3. **Known-Plaintext Attack:** If a portion of the plaintext is known, it can be used to determine the key and decrypt the entire message.

**Applications:** The Playfair Cipher, like the Vigenère Cipher, has historical significance as one of the early methods of encryption. In modern times, it is mainly used for educational purposes, teaching the concept of digraphic substitution ciphers and the construction of the Playfair Square. It is not suitable for secure communication due to its vulnerability to cryptanalysis but serves as an excellent tool for understanding cryptographic principles.

**References:**

1. **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>
2. **Wikipedia contributors. (2023)** In Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/wiki/Playfair_cipher>

**Cryptool.org:** [https://www.cryptool.org/en/](https://www.cryptool.org/en/cto/transposition)

EXPERIMENT NO – 7

**Aim:** The aim of the n-gram Hill Cipher implementation is to provide a secure and efficient way to encrypt and decrypt messages using a matrix-based encryption technique. Unlike the traditional Hill Cipher, which encrypts letters individually, the n-gram Hill Cipher works with groups of n characters (n-grams) at a time, enhancing its ability to protect against frequency analysis and other cryptanalysis methods

**Introduction:** The n-gram Hill Cipher is an advanced extension of the classical Hill Cipher, a polygraphic substitution cipher that operates on individual letters. While the Hill Cipher encrypts plaintext letter by letter, the n-gram Hill Cipher enhances security by working with groups of n characters at a time, referred to as n-grams. This modification makes the cipher more robust against various cryptanalysis techniques, particularly frequency analysis, which is often effective against simple substitution ciphers.

The concept of the Hill Cipher, whether classical or n-gram, is rooted in linear algebra and matrix mathematics. It employs a key matrix to transform the plaintext n-grams into ciphertext n-grams using matrix multiplication. The size of the key matrix and the choice of n have a significant impact on the security and complexity of the encryption.

The n-gram Hill Cipher introduces a novel layer of complexity in the encryption process, as the encryption and decryption of n-grams require the application of matrix operations and modular arithmetic. This added complexity increases the effort required for attackers to decipher the ciphertext without knowledge of the key matrix.

The flexibility of the n-gram Hill Cipher is seen in its adaptability to various n-gram sizes, making it suitable for different security requirements. Smaller n-grams may be used for faster encryption and decryption, while larger n-grams offer higher security at the expense of computational intensity.

**Program (source code):**

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

void **displayMatrix**(vector<vector<int>> mat) {

    int n = mat.**size**();

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            cout **<<** mat**[**i**][**j**]** **<<** " ";

        }

        cout **<<** **endl**;

    }

}

string **hillCipherEncrypt**(string plaintext, vector<vector<int>> keyMatrix) {

    int mod = 26; // Modulus for the alphabet size

    int n = keyMatrix.**size**();

    cout **<<** "\nKey matrix: " **<<** **endl**;

**displayMatrix**(keyMatrix);

    // Encrypt the plaintext

    int len = plaintext.**length**();

    string ciphertext = "";

    // Add 'X' if length of plaintext isnt multiple of n

    while (len % n != 0) {

        plaintext **+=** 'X';

        len++;

    }

    // Process the plaintext in blocks of size n

    for (int i = 0; i < len; i += n) {

        vector<int> **block**(n, 0); // initialize a block to store current n char

        vector<int> **result**(n, 0); // " " encrypted  n char

        // to numeric

        for (int j = 0; j < n; j++) {

            block**[**j**]** = plaintext**[**i + j**]** - 'A';

        }

        // Matrix multiplication for current block

        for (int j = 0; j < n; j++) {

            for (int k = 0; k < n; k++) {

                result**[**j**]** += keyMatrix**[**j**][**k**]** \* block**[**k**]**;

                result**[**j**]** %= mod;

            }

        }

        // convert back to Alphabets and append to result

        for (int j = 0; j < n; j++) {

            ciphertext **+=** (char)(result**[**j**]** + 'A');

        }

    }

    return ciphertext;

}

// Mod of a matrix

vector<vector<int>> **modMatrix**(vector<vector<int>> matrix, int mod) {

    int n = matrix.**size**();

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            matrix**[**i**][**j**]** = (matrix**[**i**][**j**]** % mod + mod) % mod;

        }

    }

    return matrix;

}

// Function to calculate the modular inverse of a number

int **modInverse**(int a, int m) {

    a = a % m;

    for (int x = 1; x < m; x++) {

        if ((a \* x) % m == 1)

            return x;

    }

    return -1; // Inverse DNE

}

// Function to calculate the determinant of a square matrix

int **determinant**(vector<vector<int>> matrix, int n) {

    if (n == 1) {

        return matrix**[**0**][**0**]**; // Base case

    }

    int det = 0; // Initialize

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

        vector<vector<int>> **subMatrix**(n - 1, **vector**<int>(n - 1, 0)); // Create a submatrix for each column.

        int subMatrixRow = 0, subMatrixCol = 0;

        for (int row = 1; row < n; row++) {

            // Create a submatrix by excluding the current row and column.

            subMatrixCol = 0;

            for (int subCol = 0; subCol < n; subCol++) {

                if (subCol != col) {

                    subMatrix**[**subMatrixRow**][**subMatrixCol**]** = matrix**[**row**][**subCol**]**;

                    subMatrixCol++;

                }

            }

            subMatrixRow++;

        }

        int sign = (col % 2 == 0) ? 1 : -1; // Determine the sign for this element

        // Recursively calculate the det of the submatrix and give sign

        det += sign \* matrix**[**0**][**col**]** \* **determinant**(subMatrix, n - 1);

    }

    return det;

}

// Function to obtain a submatrix by excluding a specific row and column

vector<vector<int>> **getSubMatrix**(vector<vector<int>> matrix, int excludeRow, int excludeCol) {

    int n = matrix.**size**();

    vector<vector<int>> **subMatrix**(n - 1, **vector**<int>(n - 1, 0)); // Initialize a submatrix of smaller size.

    int subMatrixRow = 0, subMatrixCol;

    for (int row = 0; row < n; row++) {

        if (row == excludeRow) {

            continue;  // Skip the excluded row.

        }

        subMatrixCol = 0;

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

            if (col != excludeCol) {

                subMatrix**[**subMatrixRow**][**subMatrixCol**]** = matrix**[**row**][**col**]**; // Copy elements to the submatrix.

                subMatrixCol++;

            }

        }

        subMatrixRow++;

    }

    return subMatrix; // Return the resulting submatrix.

}

// Function to compute the inverse of a matrix modulo mod

vector<vector<int>> **inverseMatrix**(vector<vector<int>> keyMatrix, int mod) {

    int n = keyMatrix.**size**();

    vector<vector<int>> **inverseKey**(n, **vector**<int>(n, 0)); // Initialize

    // Calculate the determinant of the key matrix.

    int det = **determinant**(keyMatrix, n);

    // Make sure to have positive value for det

    det = (det % mod + mod) % mod;

    // Calculate the modular inverse of the determinant.

    int detInverse = **modInverse**(det, mod);

    if (detInverse == -1) {

        cout **<<** "Inverse doesn't exist. Unable to decrypt." **<<** **endl**;

        return inverseKey;

    }

    // Calculate the cofactor of the key matrix.

    vector<vector<int>> **cofactor**(n, **vector**<int>(n, 0));

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            int sign = ((i + j) % 2 == 0) ? 1 : -1; // Determine the sign

            // Calculate det of submatrices for cofactors.

            vector<vector<int>> **subMatrix**(n - 1, **vector**<int>(n - 1, 0));

            subMatrix **=** **getSubMatrix**(keyMatrix, i, j);

            cofactor**[**i**][**j**]** = sign \* **determinant**(subMatrix, n - 1);

        }

    }

    // Transpose of the cofactor

    vector<vector<int>> **adjugate**(n, **vector**<int>(n, 0));

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            adjugate**[**i**][**j**]** = cofactor**[**j**][**i**]**; .

        }

    }

    // Multiply the adjugate by the modular inverse of the determinant.

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            inverseKey**[**i**][**j**]** = (adjugate**[**i**][**j**]** \* detInverse) % mod;

        }

    }

    return **modMatrix**(inverseKey, mod); // Apply modulo operation to the entire matrix.

}

// Function to decrypt

void **hillCipherDecrypt**(string ciphertext, vector<vector<int>> keyMatrix) {

    int mod = 26;

    int n = keyMatrix.**size**();

    // Calculate the inverse key matrix

    vector<vector<int>> inverseKey = **inverseMatrix**(keyMatrix, mod);

    cout **<<** "\nInverse Key matrix: " **<<** **endl**;

**displayMatrix**(inverseKey);

    int len = ciphertext.**length**();

    string plaintext = "";

    // Process the ciphertext in blocks of size n

    for (int i = 0; i < len; i += n) {

        vector<int> **block**(n, 0);

        vector<int> **result**(n, 0);

        // to numeric

        for (int j = 0; j < n; j++) {

            block**[**j**]** = ciphertext**[**i + j**]** - 'A';

        }

       // Perform matrix multiplication with the inverse key matrix for the current block

        for (int j = 0; j < n; j++) {

            for (int k = 0; k < n; k++) {

                result**[**j**]** += inverseKey**[**j**][**k**]** \* block**[**k**]**;

                result**[**j**]** %= mod;

            }

        }

        // Back to Alphabets

        for (int j = 0; j < n; j++) {

            plaintext **+=** (char)(result**[**j**]** + 'A');

        }

    }

    cout **<<** "Decrypted Text: " **<<** plaintext **<<** **endl**;

}

int **main**() {

    string message = "MEETMEATTHEPARKATNOON";

    vector<vector<int>> keyMatrix = {

        {2, 1, 0, 0, 0, 0},

        {1, 2, 1, 0, 0, 0},

        {0, 1, 2, 1, 0, 0},

        {0, 0, 1, 2, 1, 0},

        {0, 0, 0, 1, 2, 1},

        {0, 0, 0, 0, 1, 2}

    };

    // vector<vector<int>> keyMatrix = {

    //     {3,3},

    //     {2,5}

    // };

    string ciphertext = **hillCipherEncrypt**(message, keyMatrix);

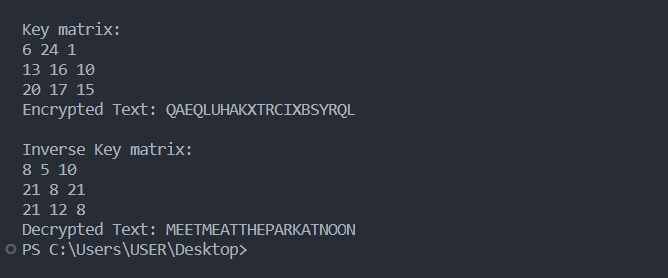
    cout **<<** "Encrypted Text: " **<<** ciphertext **<<** **endl**;

**hillCipherDecrypt**(ciphertext, keyMatrix);

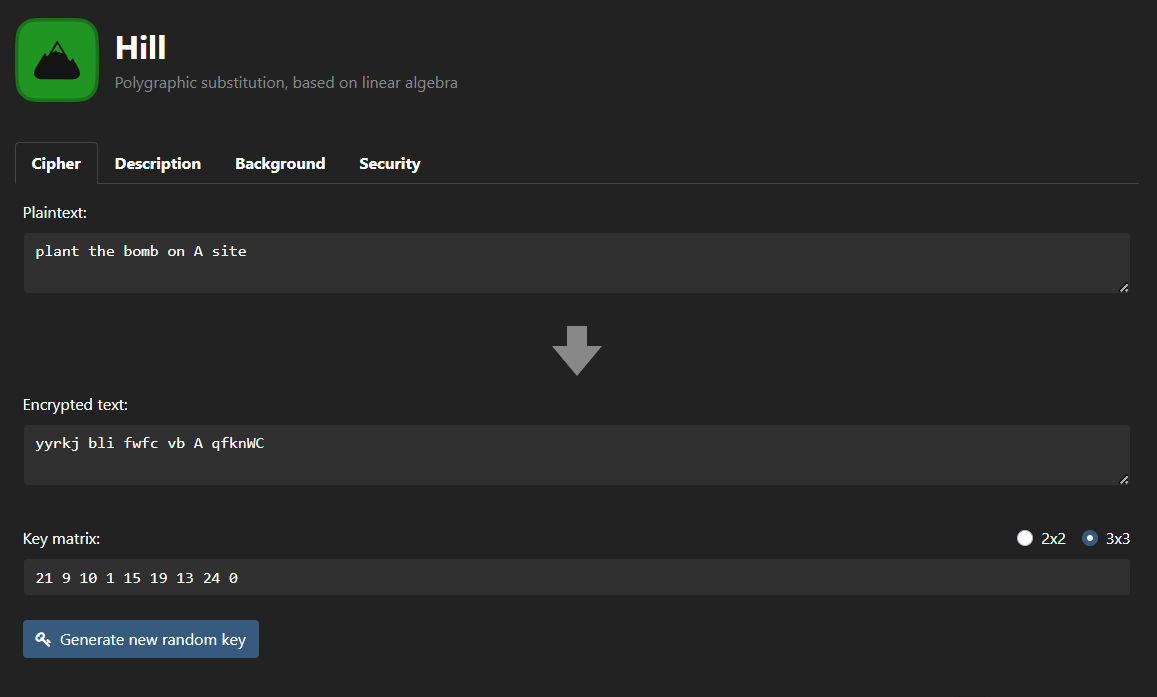
    return 0;

}

**Output (program):**

****

**Output (Cryptool):**

****

**Cryptanalysis:**

Cryptanalysis of the n-gram Hill Cipher involves attempting to break the encryption to reveal the original message without knowledge of the key. This process requires careful analysis and various techniques to exploit potential weaknesses in the cipher. Here are some methods and considerations for cryptanalysis of the n-gram Hill Cipher:

1. **Brute Force Attack**:
   * In a brute force attack, the attacker tries all possible combinations of key matrices. The number of key possibilities grows exponentially with the matrix size and the number of possible characters in the n-grams. This makes brute force less practical as the n-gram size and key matrix dimensions increase.
2. **Frequency Analysis on n-grams**:
   * Frequency analysis can still be applied, but the larger n-gram size makes it more resilient to straightforward frequency attacks. Instead of analyzing individual letters, the attacker must analyze the frequency of n-grams, which requires significantly more data and computational effort.
3. **Known-Plaintext Attack**:
   * If an attacker has access to both the plaintext and the corresponding ciphertext, they may use this knowledge to reverse-engineer the key. By comparing n-grams in the plaintext and ciphertext, they can deduce elements of the key matrix and, with enough information, potentially reconstruct the entire key.
4. **Ciphertext-Only Attack**:
   * In a ciphertext-only attack, the attacker has only the encrypted message and no knowledge of the plaintext or key. This is the most challenging scenario for cryptanalysis. The complexity of this task increases with the size of the n-grams and key matrix.
5. **Matrix Analysis**:
   * The structure of the key matrix may inadvertently reveal information about the key, especially if the matrix is not chosen securely. An attacker may look for patterns or properties that allow them to deduce portions of the key matrix.
6. **Known-Key Attack**:
   * In some cases, if an attacker can determine part of the key or guesses it accurately, they may be able to decrypt the ciphertext. For example, if they know the key matrix but not the specific key values, they can perform matrix operations to reverse the encryption.

**Applications:**

**Applications** of the n-gram Hill Cipher encompass various domains where secure encryption is required to protect data and ensure confidentiality. The use of n-grams, as opposed to individual letters, increases the cipher's resistance to frequency analysis and other cryptanalysis methods. Some of the specific applications include:

1. **Secure Messaging**:
   * The n-gram Hill Cipher is employed to encrypt messages in communication systems to ensure that sensitive information remains confidential during transmission. This is particularly important in email communication, instant messaging, and other forms of online communication where privacy is a concern.
2. **Data Protection**:
   * In sectors like finance, healthcare, and government, the n-gram Hill Cipher is used to safeguard sensitive and personal data. It is applied to encrypt financial transactions, electronic health records, and classified government communications, providing a robust layer of security.
3. **Military and Defense**:
   * Military and defense agencies utilize the n-gram Hill Cipher for the secure transmission of classified information, battle orders, and tactical data. Its ability to resist cryptanalysis ensures that sensitive military communications remain confidential.
4. **Cryptography Research**:
   * Cryptographers and researchers use the n-gram Hill Cipher as a subject of study to explore the effectiveness of n-gram-based encryption. This research contributes to the development of advanced cryptographic methods and understanding the challenges involved in breaking such ciphers.
5. **Secure File Storage**:
   * The cipher can be employed to encrypt files and data at rest, such as in cloud storage, external hard drives, and database systems. This protects files from unauthorized access and data breaches.

**References:**

1. **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>
2. **Wikipedia contributors. (2023)**: In Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/wiki/Hill_cipher>
3. **Cryptool.org:** <https://www.cryptool.org/en/cto/hill>

**EXPERIMENT-8**

**Aim:** The aim of this experiment is to explore the fundamentals of cryptographic techniques and their implementation in Python. In this experiment, we will utilize Python libraries for encryption and decryption using different block ciphers. The primary objective is to gain hands-on experience with the Crypto and cryptography libraries, understanding how they work, and comparing their functionality with a focus on symmetric key encryption. We will create Python programs for four different block ciphers: DES, Triple DES (3DES), AES-128, and Triple AES (AES-128 with triple encryption).

**Introduction:**

Cryptographic techniques play a critical role in securing data and communications in today's digital world. These techniques involve the use of encryption algorithms to protect sensitive information from unauthorized access. Python provides various libraries that enable the implementation of encryption and decryption processes, making it an excellent platform for exploring cryptography.

In this experiment, we will work with the Crypto library (for DES and Triple DES encryption) and the cryptography library (for AES encryption). Each of the four Python programs showcases a different block cipher and demonstrates encryption and decryption operations. Let's briefly introduce the four programs:

1. **Program 1: DES Encryption and Decryption**
   * In this program, we use the Crypto library to perform encryption and decryption with the Data Encryption Standard (DES) algorithm. DES is a symmetric key block cipher that has been widely used but is now considered outdated due to its small key size.
2. **Program 2: Triple DES (3DES) Encryption and Decryption**
   * Building upon the first program, we introduce Triple DES (3DES) encryption. 3DES applies the DES algorithm three times with three different keys, enhancing its security. This program showcases 3DES encryption and decryption using Python.
3. **Program 3: AES-128 Encryption and Decryption**
   * Transitioning to the cryptography library, we implement encryption and decryption with the Advanced Encryption Standard (AES). AES is a widely adopted symmetric key encryption algorithm known for its security and efficiency. This program demonstrates AES encryption and decryption with a 128-bit key.
4. **Program 4: Triple AES (AES-128 with Triple Encryption)**
   * In the final program, we continue using the cryptography library to showcase the concept of Triple AES. This program performs triple encryption with AES-128, similar to 3DES, but with the more modern AES algorithm. Triple AES offers enhanced security with three sequential AES operations.

**Program:**

1. DES :

from Crypto.Cipher import DES

from Crypto.Random import get\_random\_bytes

# Key for DES encryption/decryption (must be 8 bytes long)

key = get\_random\_bytes(8)

# Initialize the DES cipher for encryption

cipher = DES.**new**(key, DES.MODE\_ECB)

# The plaintext message to be encrypted (must be a multiple of 8 bytes)

plaintext = b'This is a the plaintext to be encrypted'

# Ensure that the plaintext is a multiple of 8 bytes by padding it if necessary

while **len**(plaintext) % 8 != 0:

    plaintext += b' '

# Encrypt the plaintext

ciphertext = cipher.**encrypt**(plaintext)

# Initialize the DES cipher for decryption

decipher = DES.**new**(key, DES.MODE\_ECB)

# Decrypt the ciphertext

decrypted\_text = decipher.**decrypt**(ciphertext)

# Remove any padding from the decrypted plaintext

decrypted\_text = decrypted\_text.**rstrip**()

# Print the results

**print**("Plaintext: ", plaintext)

**print**("Ciphertext: ", ciphertext)

**print**("Decrypted Text: ", decrypted\_text)

1. Triple DES:

from Crypto.Cipher import DES3

from Crypto.Random import get\_random\_bytes

key = get\_random\_bytes(24)

# Initialize the Triple DES cipher for encryption

cipher = DES3.**new**(key, DES3.MODE\_ECB)

plaintext = b'This is the plaintext to be encrypted'

# padding to ensure plaintext is a multiple of 8

while **len**(plaintext) % 8 != 0:

    plaintext += b' '

# encryption

ciphertext = cipher.**encrypt**(plaintext)

# Initialize the Triple DES cipher for decryption

decipher = DES3.**new**(key, DES3.MODE\_ECB)

# decryption

decrypted\_text = decipher.**decrypt**(ciphertext)

# Remove any padding from the decrypted plaintext

decrypted\_text = decrypted\_text.**rstrip**()

**print**("Plaintext: ", plaintext)

**print**("Ciphertext: ", ciphertext)

**print**("Decrypted Text: ", decrypted\_text)

1. Triple AES:

from Crypto.Cipher import AES

from Crypto.Random import get\_random\_bytes

# 32 bytes key for AES-256

key = get\_random\_bytes(32)

# Initialize the Triple AES cipher for encryption

cipher = AES.**new**(key, AES.MODE\_ECB)

plaintext = b'This is the plaintext to be encrypted'

while **len**(plaintext) % 16 != 0:

    plaintext += b' '

# encryption

ciphertext = cipher.**encrypt**(plaintext)

# Initialize the Triple AES cipher for decryption

decipher = AES.**new**(key, AES.MODE\_ECB)

# decryption

decrypted\_text = decipher.**decrypt**(ciphertext)

**print**("Plaintext: ", plaintext)

**print**("Ciphertext: ", ciphertext)

**print**("Decrypted Text: ", decrypted\_text)

1. Triple AES with 3 keys:

from Crypto.Cipher import AES

from Crypto.Random import get\_random\_bytes

# Generate three random 16-byte (128-bit) keys for Triple AES (AES-128)

key1 = get\_random\_bytes(16)

key2 = get\_random\_bytes(16)

key3 = get\_random\_bytes(16)

# Initialize the Triple AES cipher for encryption

cipher1 = AES.**new**(key1, AES.MODE\_ECB)

cipher2 = AES.**new**(key2, AES.MODE\_ECB)

cipher3 = AES.**new**(key3, AES.MODE\_ECB)

plaintext = b'This is the plaintext message to be deleted'

while **len**(plaintext) % 16 != 0:

    plaintext += b' '

# encryption

ciphertext = cipher1.**encrypt**(cipher2.**encrypt**(cipher3.**encrypt**(plaintext)))

# decryption

decrypted\_text = cipher3.**decrypt**(cipher2.**decrypt**(cipher1.**decrypt**(ciphertext)))

**print**("Plaintext: ", plaintext)

**print**("Ciphertext: ", ciphertext)

**print**("Decrypted Text: ", decrypted\_text)

**Cryptanalysis**:

Cryptanalysis is a field of study that focuses on analyzing and breaking cryptographic systems. It encompasses various techniques and approaches to understand and exploit the vulnerabilities of cryptographic algorithms, keys, and protocols. Cryptanalysts aim to discover weaknesses that could potentially compromise the security of encrypted data. The two main types of cryptanalysis are:

1. Brute Force Attacks: These involve systematically trying all possible keys or combinations to decrypt the ciphertext.
2. Cryptographic Attacks: These exploit specific vulnerabilities or weaknesses in the encryption algorithm or implementation.

**Applications :**

1. Data Protection in Communications:
   * Encryption methods like DES, Triple DES, and AES are widely used to secure data during transmission over networks. They protect sensitive information in email communication, online banking, and messaging applications, ensuring confidentiality.
2. Secure Storage of Data:
   * Encryption is essential for securing data at rest. It is used to protect files, databases, and sensitive information stored on devices, cloud servers, and storage media.
3. E-commerce and Online Transactions:
   * In online shopping and e-commerce, encryption ensures the security of financial transactions. AES is commonly used to encrypt credit card details and other sensitive payment information.
4. Secure Communication Channels:
   * Encryption is a fundamental component of secure communication channels, including VPNs and secure sockets layer (SSL)/Transport Layer Security (TLS) protocols. These methods protect data exchanged between users and websites, preventing eavesdropping .

**EXPERIMENT-9**

**Aim:** The aim of implementing the RSA (Rivest-Shamir-Adleman) algorithm is to enable secure and efficient asymmetric encryption, decryption, and digital signature generation. RSA is a cornerstone of modern cryptography, designed to provide a robust method for secure communication and data protection.

**Introduction:** The RSA (Rivest-Shamir-Adleman) Cipher stands as a widely employed cryptographic algorithm in the realm of asymmetric-key cryptography. Crafted by the trio of Ron Rivest, Adi Shamir, and Leonard Adleman in 1977, RSA adheres to an asymmetric-key structure. This implies the utilization of two distinct keys: one designated for encryption and the other for decryption.

Critical characteristics and underpinning concepts integral to the RSA Cipher encompass the following:

1. **Asymmetric Encryption**: RSA's foundation is rooted in asymmetric encryption, where it employs a dual-key system—a public key for encryption and a private key for decryption. This distinct setup ensures that messages encrypted with the public key can only be unveiled using the corresponding private key, thereby fortifying secure communication.
2. **Prime Number Operations**: RSA leans upon the mathematical properties of prime numbers. The bedrock of its security resides in the formidable challenge of factoring the product of two large prime numbers. The public key encompasses two such numbers, while the private key comprises the factors of the product derived from these prime entities.
3. **Key Generation**: The inception of RSA keys transpires through the selection of two sizeable prime numbers. The product arising from this pairing serves as the modulus for both the public and private keys. The critical exponents for encryption and decryption are then derived, their values intertwined with these prime elements and the totient function.
4. **Modular Arithmetic**: A pivotal role is played by modular arithmetic within the RSA framework. All mathematical operations operate under the umbrella of modular arithmetic, with calculations confined within the bounds of the modulus (n). This ensures results adhere to the prescribed range of [0, n-1].
5. **Public Key Encryption**: An intrinsic feature of RSA is its capacity for public key encryption. This empowers any entity to employ the public key of the intended recipient for message encryption, with the assurance that only the holder of the corresponding private key holds the decrypting capability.
6. **Digital Signatures**: RSA also excels in the realm of digital signatures, functioning as a safeguard for data integrity and authentication. The sender wields their private key to affix a signature to a message, and the recipient deploys the sender's public key to validate the signature's authenticity.
7. **Key Security**: The bedrock of RSA's security hinges on the meticulous protection and secrecy of the private key. The significance of guarding the private key cannot be overstated, as any compromise could lead to the decryption of all messages encrypted using the corresponding public key.
8. **Key Length**: RSA's security correlates directly with key length. Longer keys, such as those spanning 1024, 2048, or 4096 bits, bolster security but, in turn, demand heightened computational resources.

This encapsulation underscores the core principles and attributes of the RSA Cipher, illuminating its indispensable role in contemporary cryptography, ensuring secure and confidential communication in the interconnected digital landscape.

**Program & Output:**

from math import **sqrt**

from random import randint as rand

import random

# Euclidean Algorithm to find the greatest common divisor (GCD)

def **gcd**(a, b):

    if b == 0:

        return a

    else:

        return **gcd**(b, a % b)

# Calculate the modular multiplicative inverse

def **mod\_inverse**(a, m):

    for x in range(1, m):

        if (a \* x) % m == 1:

            return x

    return -1

# Check if a number is prime

def **isprime**(n):

    if n < 2:

        return False

    elif n == 2:

        return True

    else:

        for i in range(2, int(**sqrt**(n)) + 1, 2):

            if n % i == 0:

                return False

    return True

# Generate random prime numbers p and q

p = rand(1, 1000)

q = rand(1, 1000)

# Generate an RSA key pair (public key and private key)

def **generate\_keypair**(p, q, keysize):

    nMin = 1 << (keysize - 1)

    nMax = (1 << keysize) - 1

    primes = [2]

    start = 1 << (keysize // 2 - 1)

    stop = 1 << (keysize // 2 + 1)

    if start >= stop:

        return []

    for i in range(3, stop + 1, 2):

        for p in primes:

            if i % p == 0:

                break

        else:

            primes.**append**(i)

    while primes and primes[0] < start:

        del primes[0]

    while primes:

        p = random.choice(primes)

        primes.**remove**(p)

        q\_values = [q for q in primes if nMin <= p \* q <= nMax]

        if q\_values:

            q = random.choice(q\_values)

            break

    n = p \* q

    phi = (p - 1) \* (q - 1)

    e = random.randrange(1, phi)

    g = **gcd**(e, phi)

    while True:

        e = random.randrange(1, phi)

        g = **gcd**(e, phi)

        d = **mod\_inverse**(e, phi)

        if g == 1 and e != d:

            break

    return ((e, n), (d, n))

# Encrypt a message using the public key

def **encrypt**(msg\_plaintext, package):

    e, n = package

    msg\_ciphertext = [**pow**(**ord**(c), e, n) for c in msg\_plaintext]

    return msg\_ciphertext

# Decrypt a message using the private key

def **decrypt**(msg\_ciphertext, package):

    d, n = package

    msg\_plaintext = [**chr**(**pow**(c, d, n)) for c in msg\_ciphertext]

    return ''.**join**(msg\_plaintext)

bit\_length = 4

msg = 'PlantTheBomb'

# Generate RSA key pair with p, q, and specified key length

public, private = **generate\_keypair**(p, q, 2\*\*bit\_length)

**print**("Public Key: ", public)

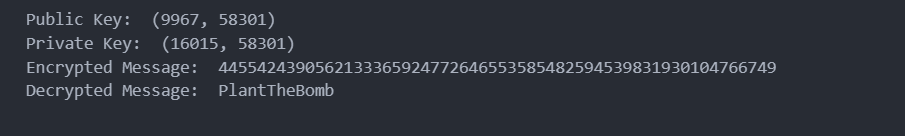
**print**("Private Key: ", private)

# Encrypt and then decrypt a message

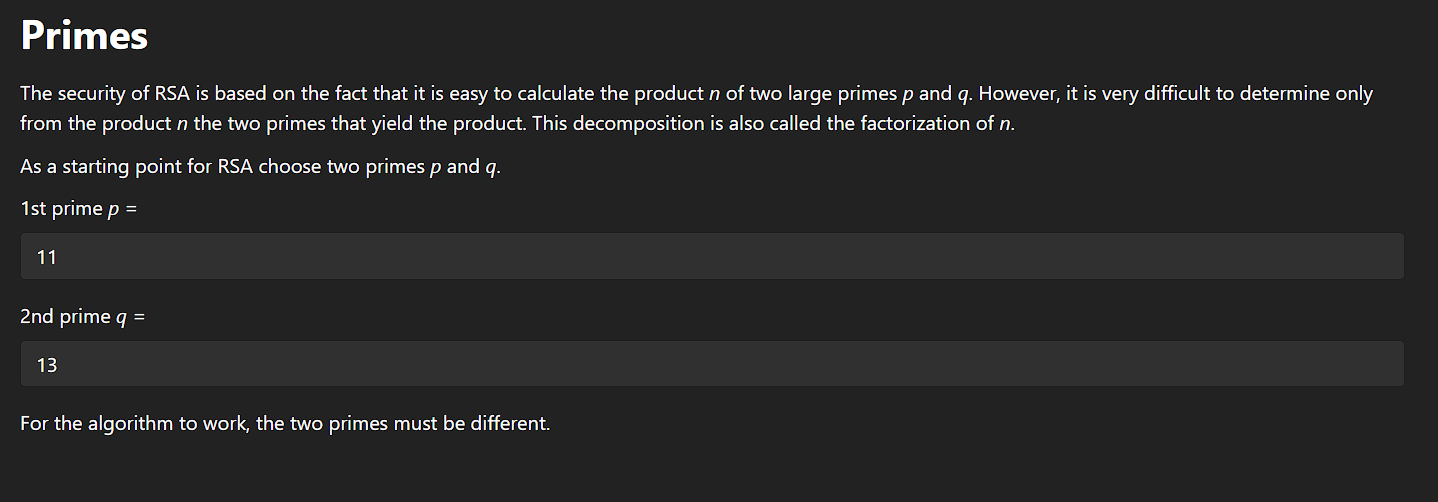
encrypted\_msg = **encrypt**(msg, public)

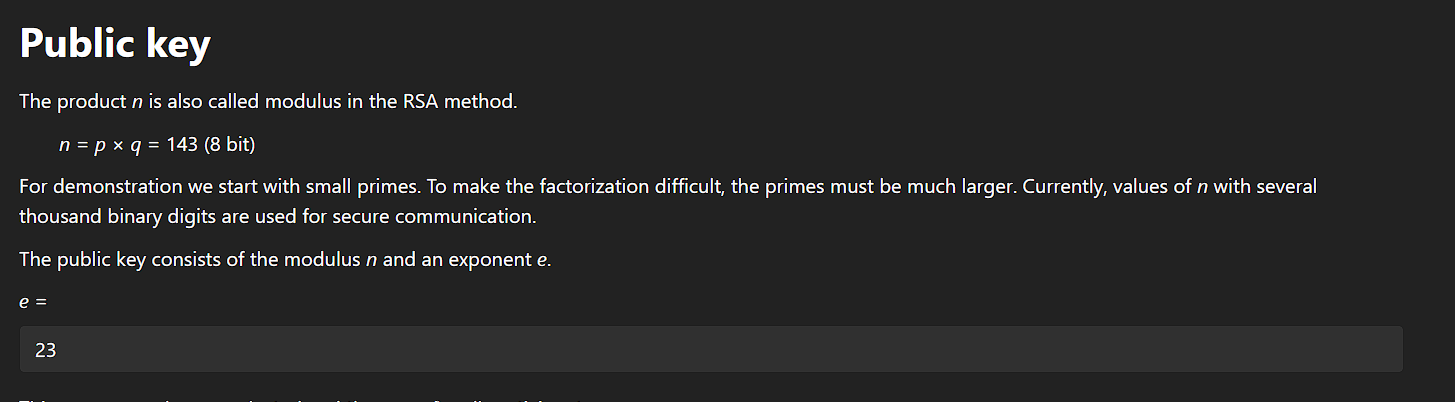
**print**("Encrypted Message: " , ''.**join**(map(lambda x: str(x), encrypted\_msg)))

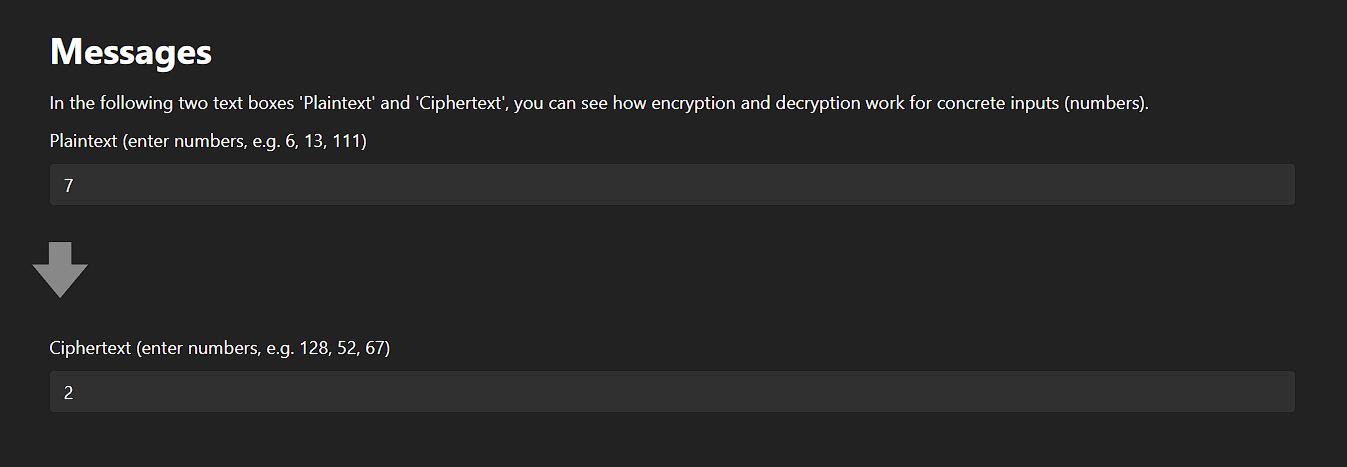
**print**("Decrypted Message: " , **decrypt**(encrypted\_msg, private))

**Output:  
  
**

**Cryptool:**

****

****

****

**Cryptanalysis:**

Cryptanalysis of the RSA (Rivest-Shamir-Adleman) algorithm involves attempting to break the encryption, digital signatures, or key exchange processes by exploiting weaknesses in the mathematical foundations or the way it is implemented. RSA is considered a robust and widely trusted cryptographic algorithm, and successful cryptanalysis typically requires overcoming significant computational challenges. Nevertheless, several potential attacks and vulnerabilities should be considered:

1. **Brute Force Attack:**
   * A brute force attack involves systematically trying all possible private key combinations by attempting to factor the modulus, N, into its two prime factors, p and q. The security of RSA depends on the difficulty of factoring large semiprime numbers. With advances in computing power, larger key sizes have become necessary to resist this type of attack.
2. **Integer Factorization:**
   * The primary threat to RSA security is integer factorization. If an attacker can efficiently factor the modulus, N, into its prime factors (p and q), they can compute the private key. Factoring large numbers remains a computationally intensive task, and RSA security relies on the use of large prime numbers.
3. **Timing Attacks:**
   * Timing attacks exploit variations in the time taken to execute certain RSA operations. By measuring these variations, an attacker can gain information about the private key. Implementing constant-time operations can mitigate this type of attack.
4. **Chosen-Plaintext Attacks:**
   * In a chosen-plaintext attack, an attacker encrypts plaintext of their choice and observes the corresponding ciphertext. By examining the relationship between plaintext and ciphertext, the attacker may gain information about the private key. Padding schemes are used to counteract this threat.
5. **Common Modulus Attack:**
   * If two different entities use the same modulus N and have different public exponents, it may be possible for an attacker with knowledge of both public exponents and their corresponding ciphertext to recover the plaintext of both parties.

**Applications:**

Despite the potential vulnerabilities and cryptanalysis techniques, the RSA Cipher remains one of the most widely used and trusted encryption methods for securing data in various applications:

1. **Secure Communication:** RSA is a fundamental building block for secure communication over the internet. It is used for secure email communication, SSL/TLS encryption in web browsers, and secure file transfers.
2. **Digital Signatures:** RSA is employed in digital signatures to verify the authenticity and integrity of digital documents and messages. It plays a crucial role in secure online transactions and certificate authorities.
3. **Secure Key Exchange:** RSA is often used for secure key exchange protocols, such as the Diffie-Hellman key exchange, to establish a secure session key between parties.
4. **Data Encryption:** RSA is used for encrypting sensitive data at rest and in transit. It is a core component of encryption standards like PGP (Pretty Good Privacy) and S/MIME.

**References:**

1. **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>
2. **Wikipedia contributors. (2023)**: In Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/wiki/RSA_(cryptosystem)>
3. **Cryptool.org:** <https://www.cryptool.org/en/cto/rsa-step-by-step>

**Experiment-10**

**Aim**:

The aim of the experiment is to demonstrate the generation and verification of digital signatures using the RSA (Rivest-Shamir-Adleman) algorithm for file integrity and authenticity. The primary objectives of this experiment include:

1. **Digital Signature Generation**: Generate a digital signature for a selected file using a private key. The digital signature should serve as proof of the file's authenticity and integrity.
2. **Digital Signature Verification**: Verify the generated digital signature using the corresponding public key. Successful verification ensures that the file has not been tampered with and that it was indeed signed by the private key holder.
3. **Data Integrity and Security**: Showcase how digital signatures enhance data security by detecting unauthorized changes or alterations in files. This process is vital for ensuring the integrity and authenticity of digital information.
4. **Application of RSA in Cryptography**: Illustrate the practical application of the RSA algorithm in creating and validating digital signatures, a critical component of secure communication, document verification, and data protection.

**Introduction:**

RSA (Rivest-Shamir-Adleman) is a widely used asymmetric cryptographic algorithm that serves multiple purposes in secure communication. In the context of RSA, two key models are commonly applied: sender's authentication only and message confidentiality along with sender's authentication. These models offer different levels of security and address distinct aspects of secure communication.

**1. Sender's Authentication Only:**

In the first model, RSA is employed to provide sender's authentication. When a sender wishes to prove their identity to a recipient, they digitally sign a message using their private key. This process generates a digital signature unique to the sender. Upon receiving the message, the recipient verifies the sender's authenticity by decrypting the signature with the sender's public key. If the decrypted signature matches the content, the recipient can be confident that the message indeed originates from the claimed sender. This model is effective in ensuring the sender's identity but does not provide message confidentiality.

**2. Message Confidentiality and Sender's Authentication:**

The second model utilizes RSA to achieve both message confidentiality and sender's authentication. Here, the sender signs the message with their private key to establish their identity. Subsequently, the sender encrypts the entire message, including the sender's signature, using the recipient's public key. This double-layered approach ensures message confidentiality because only the recipient with the corresponding private key can decrypt and read the message. In addition to confidentiality, the recipient can decrypt the message, validate the sender's signature using the sender's public key, and thus ensure the sender's authenticity. This method is a secure means of exchanging confidential messages while upholding the sender's credibility in the communication process. It combines the benefits of data privacy and sender authentication for comprehensive security in message transmission.

**Program & Output:**

# Authentication through Digital Signatures

# Digital Signature Generation (Sender)

def **sign\_message**(msg\_plaintext, sender\_private\_key):

    d, n = sender\_private\_key

    signature = [**pow**(**ord**(c), d, n) for c in msg\_plaintext]

    return signature

# Digital Signature Verification (Receiver)

def **verify\_signature**(signature, msg\_plaintext, sender\_public\_key):

    e, n = sender\_public\_key

    decrypted\_signature = [**pow**(s, e, n) for s in signature]

    decrypted\_msg = ''.**join**([**chr**(d) for d in decrypted\_signature])

    return decrypted\_msg == msg\_plaintext

# Sender's Key Pair

sender\_private\_key = private

sender\_public\_key = public

# Message to be sent

message\_to\_send = "PlantTheBombOnAsite"

# Sender signs the message

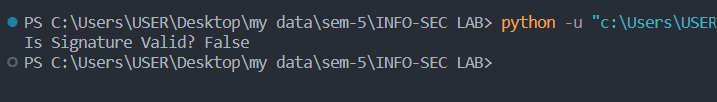
signature = **sign\_message**(message\_to\_send, sender\_private\_key)

message = "BombPlantedOnAsite"

# Receiver verifies the sender's signature

is\_signature\_valid = **verify\_signature**(signature, message, sender\_public\_key)

**print**("Is the Signature Valid?", is\_signature\_valid)



# Ensuring Message Confidentiality and Sender's Authentication

# Recipient's Key Pair

recipient\_keypair = **generate\_keypair**(p, q, 2\*\*bit\_length)  # Generate recipient's key pair (replace with actual keys)

recipient\_public\_key = recipient\_keypair[0]

# Encryption (Recipient)

def **encrypt\_message**(msg\_plaintext, recipient\_public\_key):

    e, n = recipient\_public\_key

    msg\_ciphertext = [**pow**(**ord**(c), e, n) for c in msg\_plaintext]

    return msg\_ciphertext

# Decryption (Recipient)

def **decrypt\_message**(msg\_ciphertext, recipient\_private\_key):

    d, n = recipient\_private\_key

    msg\_plaintext = ''.**join**([**chr**(**pow**(c, d, n)) for c in msg\_ciphertext])

    return msg\_plaintext

# Sender's Key Pair

sender\_private\_key = sender\_private\_key  # Replace with the actual sender's private key

sender\_public\_key = sender\_public\_key    # Replace with the actual sender's public key

recipient\_private\_key = recipient\_keypair[1]

message\_to\_send = "PlantBombOnASite"

signature = **sign\_message**(message\_to\_send, sender\_private\_key)

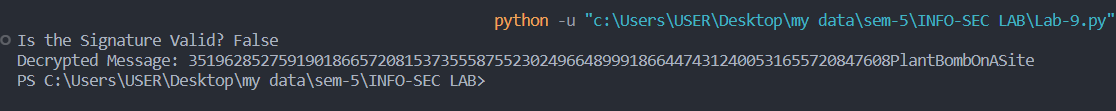
encrypted\_message = **encrypt\_message**(''.**join**(map(lambda x: str(x), signature)) + message\_to\_send, recipient\_public\_key)

decrypted\_message = **decrypt\_message**(encrypted\_message, recipient\_private\_key)

is\_signature\_valid = **verify\_signature**([int(x) for x in decrypted\_message[:bit\_length]], decrypted\_message[bit\_length:], sender\_public\_key)

**print**("Is the Signature Valid?", is\_signature\_valid)

**print**("Decrypted Message:", decrypted\_message)



**References:**

1. **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>
2. **Wikipedia contributors. (2023)**: In Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/wiki/RSA_(cryptosystem)>
3. **Cryptool.org:** <https://www.cryptool.org/en/cto/rsa-step-by-step>