Name : Ritesh Pawar
PRN : 2020BTECS00068

Assignment No 1

Cryptography and Network Security Lab (5CS453)

Name : Ritesh Pawar

PRN:2020BTECS00068

Title:

Encryption and Decryption using Ceaser Cipher.

Aim:

To Study and Implement Encryption and Decryption using Ceaser Cipher

Theory:

- Caesar Cipher, also known as the Shift Cipher, is one of the simplest and oldest encryption techniques used to secure information.
- It's a type of substitution cipher where each letter in the plain text is shifted a certain number of places down or up the alphabet.
- The number of positions a letter is shifted is determined by a key.

Encryption:

In Encryption, input is a Plain text and output is a Cipher text.

Step 1: Choose a secret key (a positive integer).
Step 2: Take the plaintext message you want to encrypt.
Step 3: Shift each letter in the message forward in the alphabet by the key positions.
Step 4: Non-alphabetical characters remain unchanged.
Step 5: The result is the ciphertext, the encrypted message.

Decryption:

In Decryption, input is a Cipher text and output is a Plain text.

Step 1: Have the same key used for encryption.
Step 2: Take the ciphertext (the encrypted message).
Step 3: Shift each letter in the ciphertext backward in the alphabet by the key positions.
Step 4: Non-alphabetical characters remain unchanged.
Step 5: The result is the plaintext, the original message.

Code:

Name : Ritesh Pawar
PRN : 2020BTECS00068

```cpp
#include <iostream>
#include <string>

using namespace std;

// Function to encrypt a message using the Caesar cipher
string encryptCaesarCipher(const string& message, int shift) {
string encryptedMessage = "";

for (char c : message) {
if (isalpha(c)) {
char base = islower(c) ? 'a' : 'A';
encryptedMessage += static_cast<char>((c - base + shift) % 26 + base);
} else {
// Preserve non-alphabetical characters
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
encryptedMessage += c;
```

```
}

}


return encryptedMessage;

}


// Function to decrypt a message encrypted
with the Caesar cipher

string decryptCaesarCipher(const string&
message, int shift) {

// To decrypt, use the negative of the shift
value

return encryptCaesarCipher(message, -shift);

}


int main() {
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
string message;
int shift;


cout << "Enter a message: ";
```

```
getline(cin, message);


cout << "Enter the shift value (positive or
negative integer): ";
cin >> shift;


string encryptedMessage =
encryptCaesarCipher(message, shift);
string decryptedMessage =
decryptCaesarCipher(encryptedMessage, shift);


cout << "Encrypted message: " <<
encryptedMessage << endl;
cout << "Decrypted message: " <<
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
decryptedMessage << endl;



return 0;

}
```

Output:

Limitations of the Caesar Cipher:

Weak Security: Caesar cipher is highly vulnerable to brute-force attacks. Since there are only 25 possible shift values (excluding the no-shift case), an attacker can easily try all possibilities to decrypt the message.

Lack of Key Management: In real-world cryptography, secure key management is crucial. In the Caesar cipher, the key is simply an integer representing the shift value. If an attacker discovers the key, they can decrypt all messages encrypted with that key.

Limited Alphabet: The Caesar cipher only works with alphabetic characters. It doesn't handle numbers, punctuation, or any other symbols, which limits its practicality for encoding various types of data.

Frequency Analysis: Even without knowing the key, attackers can still employ frequency analysis. In many languages, certain letters occur more frequently than others (e.g., 'E' is the most common letter in English). An attacker can analyze the frequency of letters in the ciphertext and make educated guesses about the key.

Name : Ritesh Pawar
PRN : 2020BTECS00068

Fixed Shift Value: The Caesar cipher relies on a fixed shift value. This makes it predictable and insecure. More advanced ciphers use variable keys, making them much more secure.

Not Suitable for Modern Cryptography: The Caesar cipher is a historical curiosity and not suitable for securing modern communications. It is easily breakable using modern computational power and cryptographic analysis techniques.

Name : Ritesh Pawar
PRN : 2020BTECS00068

Assignment No 2

Cryptography and Network Security Lab (5CS453)

Name: Ritesh Pawar

PRN: 2020BTECS00068

Title:

Encryption and Decryption using Transposition Cipher Technique.

Aim:

To Study and Implement Encryption and Decryption using Rail Fence Transposition Cipher Technique and Columnar Transposition Cipher Technique

Theory:

1. Rail Fence Transposition Cipher Technique

The Rail Fence Transposition Cipher, also known as the Zigzag Cipher, is a simple columnar transposition cipher technique.

It involves arranging the plaintext characters in a zigzag pattern across multiple rows, known as "rails," and then reading them off row by row to create the encrypted message.

While this cipher is easy to understand and implement, it lacks strong security and is mainly used for educational purposes or simple puzzles.

Encryption:

- Choose the number of rails (rows) for the zigzag pattern.
- Write the message diagonally across the rails, moving up and down.
- Read the characters row by row to form the encrypted message.

Decryption:

- Create the zigzag pattern with the chosen number of rails.
- Leave blank spaces in the pattern for characters to be placed.
- Fill in the blanks with the encrypted characters, row by row.
- Read the characters diagonally to retrieve the original message.

Advantages:

- Easy to understand and implement.
- Provides basic encryption and breaks up character repetition.

Name : Ritesh Pawar
PRN : 2020BTECS00068

Disadvantages:

- Not secure against modern cryptanalysis.
- Security depends on the number of rails, making it less practical for strong encryption.

Limitations :

Low Security: The Rail Fence Cipher is not very secure and can be easily cracked using basic cryptanalysis techniques. It provides minimal protection against modern encryption-breaking methods.

Predictable Pattern: The encryption pattern is easily recognizable, making it susceptible to pattern analysis. An attacker can often discern the fact that a Rail Fence Cipher has been used, and this knowledge can simplify decryption.

Limited Key Space: The key space of the Rail Fence Cipher is limited to the number of rails or rows used for encryption. Once an attacker identifies the number of rails, they can easily decrypt the message.

Difficulty with Long Messages: For long messages, the Rail Fence Cipher can become unwieldy, as the number of rails must be specified. If you don't know the correct number of rails, decryption becomes challenging.

Data Loss and Padding: To create a perfect zigzag pattern, you may need to add padding characters to the plaintext. These padding characters must be removed during decryption, which can complicate the process.

CODE:

```
#include <iostream>

#include <string>

#include <vector>



using namespace std;
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
// Function to encrypt a message using the
Rail Fence Cipher
string encryptRailFence(const string&
message, int rails) {
vector<string> fence(rails, "");
int currentRail = 0;
bool goingDown = false;

for (char c : message) {
fence[currentRail] += c;

if (currentRail == 0 || currentRail == rails
- 1) {
```

```
goingDown = !goingDown;
}

currentRail += goingDown ? 1 : -1;
}
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
string encryptedMessage = "";
for (const string& rail : fence) {
encryptedMessage += rail;
}

return encryptedMessage;
}

// Function to decrypt a message encrypted
with the Rail Fence Cipher
string decryptRailFence(const string&
message, int rails) {
vector<string> fence(rails, "");
vector<int> railSizes(rails, 0);
```

```
int currentRail = 0;
bool goingDown = false;
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
for (int i = 0; i < message.length(); i++) {
railSizes[currentRail]++;

if (currentRail == 0 || currentRail == rails
- 1) {
goingDown = !goingDown;
}

currentRail += goingDown ? 1 : -1;
}

int messageIndex = 0;
for (int rail = 0; rail < rails; rail++) {
fence[rail] = message.substr(messageIndex,
railSizes[rail]);
messageIndex += railSizes[rail];
}
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
string decryptedMessage = "";

currentRail = 0;

goingDown = false;


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

decryptedMessage += fence[currentRail][0];

fence[currentRail].erase(0, 1);


if (currentRail == 0 || currentRail == rails
- 1) {

goingDown = !goingDown;

}


currentRail += goingDown ? 1 : -1;

}


return decryptedMessage;

}
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
int main() {
string message;
int rails;

cout << "Enter a message: ";
getline(cin, message);

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

string encryptedMessage =
encryptRailFence(message, rails);
string decryptedMessage =
decryptRailFence(encryptedMessage, rails);

cout << "Encrypted message: " <<
encryptedMessage << endl;
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
cout << "Decrypted message: " <<
decryptedMessage << endl;



return 0;

}
```

OUTPUT:

Name : Ritesh Pawar
PRN : 2020BTECS00068

2. Columnar Transposition Cipher Technique

The Columnar Transposition Cipher is a more advanced transposition cipher technique that involves reordering the characters of a message based on a chosen keyword or key phrase.

It provides a higher level of security compared to simpler ciphers like the Rail Fence Cipher. Here's how the Columnar Transposition Cipher works:

Encryption:

- Choose a keyword or key phrase. The unique characters of the keyword determine the order of columns in the transposition grid.
- Write the message row by row into a grid, using the keyword to determine the order of columns.
- Read the characters column by column to obtain the encrypted message.

Decryption:

- Use the keyword to determine the order of columns in the transposition grid.
- Write the encrypted message into the grid column by column.
- Read the characters row by row to retrieve the original plaintext.

Advantages:

- Offers stronger security compared to simpler ciphers.
- Security depends on the length and uniqueness of the keyword.

Disadvantages:

- Can be vulnerable to attacks if the keyword is short or easily guessed.
- May require additional padding characters for messages that don't fit evenly into the grid.

Limitations :

Frequency Analysis: The Columnar Transposition Cipher retains the frequency distribution of letters in the plaintext. Attackers can still perform frequency analysis to guess the key length or potentially discover patterns within the ciphertext.

Known-Plaintext Attack: If an attacker has access to some portions of the plaintext and the corresponding ciphertext, they can use this information to make educated guesses about the key or deduce parts of the message, potentially compromising security.

Key Length Requirement: Choosing an appropriate key length can be challenging. A too-short key may not provide sufficient security, while a too-long key can make the process cumbersome, especially if you have to remember or share it securely.

Name : Ritesh Pawar
PRN : 2020BTECS00068

Limited Complexity: The Columnar Transposition Cipher, like other transposition ciphers, provides relatively low complexity in terms of encryption, making it vulnerable to automated attacks and modern cryptographic techniques.

No Substitution of Characters: Unlike substitution ciphers (e.g., Caesar Cipher), the Columnar Transposition Cipher does not alter the characters themselves, making it susceptible to known-plaintext attacks and character frequency analysis.

CODE:

```
// CPP program for illustrating

// Columnar Transposition Cipher

#include<bits/stdc++.h>

using namespace std;


// Key for Columnar Transposition

string const key = "HACK";

map<int,int> keyMap;


void setPermutationOrder()

{

// Add the permutation order into map

for(int i=0; i < key.length(); i++)
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
{

keyMap[key[i]] = i;

}

}


// Encryption
```

```
string encryptMessage(string msg)

{

int row,col,j;

string cipher = "";

/* calculate column of the matrix*/

col = key.length();

/* calculate Maximum row of the matrix*/

row = msg.length()/col;

if (msg.length() % col)

row += 1;
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
char matrix[row][col];


for (int i=0,k=0; i < row; i++)
{
for (int j=0; j<col; )
{
if(msg[k] == '\0')
{
```

```
/* Adding the padding character '_' */
matrix[i][j] = '_';
j++;
}
if( isalpha(msg[k]) || msg[k]==' ')
{
/* Adding only space and alphabet into
matrix*/
matrix[i][j] = msg[k];
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
j++;

}

k++;

}

}


for (map<int,int>::iterator ii =
keyMap.begin(); ii!=keyMap.end(); ++ii)

{

j=ii->second;
```

```
// getting cipher 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 += matrix[i][j];

}
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
}


return cipher;

}


// Decryption

string decryptMessage(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++)
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
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=keyMap.begin(); ii!=keyMap.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=keyMap.begin();

int k = 0;
```

```
for (int l=0,j; key[l]!='\0'; k++)

{

j = keyMap[key[l++]];
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
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;
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
}


// Driver Program
int main(void)
{
/* message */
string msg;
cout<<"Enter message: ";
getline(cin, msg);
setPermutationOrder();
// Calling encryption function
string cipher = encryptMessage(msg);
cout << "\nEncrypted Message: " << cipher <<
endl;
// Calling Decryption function
cout << "Decrypted Message: " <<
decryptMessage(cipher) << endl;
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
return 0;

}
```

OUTPUT:

Name : Ritesh Pawar
PRN : 2020BTECS00068

Cryptography and Network Security Lab

Name:Ritesh Pawar

PRN: 2020BTECS00068

Batch: B5

PLAY FAIR CIPHER ALGORITHM

Aim:

To encrypt plain text using PlayFair cipher and decrypt the cipher text to plain text.

Theory:

Playfair cipher is a manual symmetric encryption technique and was first diagram substitution cipher. In playfair cipher group of letters is encrypted instead of a single letter so it is little bit complex than caesar cipher. So it is hard to break playfair cipher algorithm as in simple caesar cipher one can easily predict k value and decrypt the text easily. So this playfair cipher algorithm is more secure than caesar cipher.

Code:

```
#include <bits/stdc++.h>

using namespace std;


class PlayfairCipher {

public:

static pair<vector<vector<char>>,

unordered_map<char, pair<int, int>>>

getKeyMatrixAndPositions(const string &key)
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
{
```

```
vector<vector<char>> keyMatrix(5,
vector<char>(5));
int i = 0, j = 0;
unordered_set<char> set;
unordered_map<char, pair<int, int>>
position;

for (char c : key) {
if (c == 'j')
c = 'i';

if (set.find(c) != set.end())
continue;

set.insert(c);
keyMatrix[i][j] = c;
position[c] = {i, j};
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
j++;

if (j == 5) {

j = 0;
```

```
i++;

}

}


for (char c = 'a'; c <= 'z'; c++) {

if (c == 'j')

continue;


if (set.find(c) != set.end())

continue;


set.insert(c);

keyMatrix[i][j] = c;

position[c] = {i, j};
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
j++;
if (j == 5) {
j = 0;
i++;
}
}
```

```
position[j] = position[i];

return {keyMatrix, position};
}

static vector<string> getDiagrams(const
string &text) {
int n = text.size();
int i = 0;
vector<string> diagrams;

while (i + 1 < n) {
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
if (text[i] != text[i + 1]) {

string d;

d += tolower(text[i]);

d += tolower(text[i + 1]);

diagrams.push_back(d);


i += 2;

} else {
```

```
string d;

d += tolower(text[i]);

d += 'x';

diagrams.push_back(d);


i++;

}

}


if (i == n - 1) {
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
string d;

d += tolower(text[i]);

d += 'x';

diagrams.push_back(d);

}


return diagrams;

}


static string encrypt(const string
&plaintext, const string &key) {
```

```
auto p = getKeyMatrixAndPositions(key);

auto keyMatrix = p.first;

auto position = p.second;

vector<string> diagrams =
getDiagrams(plaintext);

stringstream ciphertext;


for (string &diagram : diagrams) {
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
auto p1 = position[diagram[0]];

auto p2 = position[diagram[1]];

auto i0 = p1.first, j0 = p1.second;

auto i1 = p2.first, j1 = p2.second;


if (i0 == i1) {

diagram[0] = keyMatrix[i0][(j0 + 1) % 5];

diagram[1] = keyMatrix[i0][(j1 + 1) % 5];

} else if (j0 == j1) {

diagram[0] = keyMatrix[(i0 + 1) % 5][j0];

diagram[1] = keyMatrix[(i1 + 1) % 5][j0];

} else {

diagram[0] = keyMatrix[i0][j1];
```

```
diagram[1] = keyMatrix[i1][j0];

}


ciphertext << diagram;

}
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
string answer = ciphertext.str();

transform(answer.begin(), answer.end(),
answer.begin(), ::toupper);
return answer;
}

static string decrypt(const string
&ciphertext, const string &key) {
auto p = getKeyMatrixAndPositions(key);
auto keyMatrix = p.first;
auto position = p.second;

vector<string> diagrams =
getDiagrams(ciphertext);
```

```
stringstream plaintext;

for (string &diagram : diagrams) {
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
auto p1 = position[diagram[0]];
auto p2 = position[diagram[1]];
auto i0 = p1.first, j0 = p1.second;
auto i1 = p2.first, j1 = p2.second;


if (i0 == i1) {
diagram[0] = keyMatrix[i0][(j0 - 1 + 5) %
5];
diagram[1] = keyMatrix[i0][(j1 - 1 + 5) %
5];
} else if (j0 == j1) {
diagram[0] = keyMatrix[(i0 - 1 + 5) %
5][j0];
diagram[1] = keyMatrix[(i1 - 1 + 5) %
5][j0];
} else {
diagram[0] = keyMatrix[i0][j1];
diagram[1] = keyMatrix[i1][j0];
}
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
plaintext << diagram;
}

return plaintext.str();
}
};

int main() {
cout << "PlayFair Cipher:\n"
<< "Enter your choice:\n"
<< "1. Encrypt\n"
<< "2. Decrypt\n";

int choice;
cin >> choice;

switch (choice) {
case 1: {
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
cout << "Enter plaintext: ";
string plaintext;
```

```
cin.get();
getline(cin, plaintext);
plaintext.erase(remove_if(plaintext.begin(),
plaintext.end(), ::isspace),
plaintext.end());

cout << "Enter key : ";
string key;
cin >> key;

string ciphertext =
PlayfairCipher::encrypt(plaintext, key);

cout << "Plaintext: " << plaintext << "\n"
<< "Ciphertext: " << ciphertext << "\n";
} break;
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
case 2: {

cout << "Enter ciphertext: ";

string ciphertext;

cin >> ciphertext;
```

```


cout << "Enter key : ";

string key;

cin >> key;


string plaintext =

PlayfairCipher::decrypt(ciphertext, key);


cout << "Ciphertext: " << ciphertext << "\n"

<< "Plaintext: " << plaintext << "\n";

} break;

}


return 0;
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
}
```

Output:

Limitations:

Key Length and Reusability: The Playfair Cipher requires a relatively long and complex key, which can be challenging to manage and remember securely. Reusing the same key for multiple messages can compromise security.

Frequency Analysis: Although the Playfair Cipher can disguise single-letter frequencies, it does not effectively disguise digraph (two-letter) frequencies. Skilled cryptanalysts can use frequency analysis to break the cipher, especially when the message is long.

Known-Plaintext Attacks: If an attacker has access to both the ciphertext and corresponding plaintext for a portion of the message, they can potentially deduce parts of the key or plaintext, compromising the entire message's security.

Determined Attackers: With enough ciphertext, skilled attackers can employ various techniques like hill climbing, genetic algorithms, and simulated annealing to break the Playfair Cipher through brute force or optimization.

Pattern Recognition: The Playfair Cipher retains certain patterns in the plaintext, such as repeated letters in the same pair, which can be exploited by cryptanalysts to guess parts of the key or plaintext.

Name : Ritesh Pawar
PRN : 2020BTECS00068

Complexity for Manual Encryption: When used manually, the Playfair Cipher can be error-prone and complex, leading to mistakes in key setup or encryption/decryption.

Block Size Limitations: The Playfair Cipher encrypts plaintext in pairs of letters, which may result in uneven block sizes for messages with an odd number of characters. Padding may be needed, introducing complexity.

Not Suitable for Modern Secure Communication: Due to its weaknesses and vulnerability to modern cryptographic attacks, the Playfair Cipher is not suitable for securing sensitive or confidential information in today's digital age. Modern encryption algorithms like AES (Advanced Encryption Standard) are more secure and recommended for secure communication.

Name : Ritesh Pawar
PRN : 2020BTECS00068

Cryptography and Network Security Lab

Name: Ritesh Pawar

PRN: 2020BTECS00068

Batch: B5

VIGENERE ALGORITHM

Aim:

To encrypt plain text using vigenere cipher and convert cipher text into plain text by decryption.

Theory:

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

Code:

```
#include<bits/stdc++.h>
using namespace std;


int main()
{
  int choice;
  cout << "Choose an option:\n";
  cout << "1. Encryption\n";
  cout << "2. Decryption\n";
  cout << "Enter your choice (1 or 2): ";
  cin >> choice;
  cin.ignore(); // Clear the newline character from the input buffer


  if (choice == 1)
  {
    // Encryption
    string plainText, key, cipherText;


    cout << "\nEnter plain text: ";
    getline(cin, plainText);
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
    cout << "\nEnter key: ";
    getline(cin, key);

    // Removing spaces and converting to lowercase from plaintext
```

```
    string temp = "";
    for (int i = 0; i < plainText.size(); i++)
    {
      if (plainText[i] != ' ')
        temp += plainText[i];
    }
    plainText = temp;

    for (int i = 0; i < plainText.size(); i++)
    {
      if (plainText[i] >= 'A' && plainText[i] <= 'Z')
        plainText[i] += 32; // Convert to lowercase
    }

    // Removing spaces and converting to lowercase from key
    string temp2 = "";
    for (int i = 0; i < key.size(); i++)
    {
      if (key[i] != ' ')
        temp2 += key[i];
    }
    key = temp2;

    for (int i = 0; i < key.size(); i++)
    {
      if (key[i] >= 'A' && key[i] <= 'Z')
        key[i] += 32; // Convert to lowercase
    }

    // Encryption
    for (int i = 0; i < plainText.size(); i++)
    {
      int val = plainText[i] - 'a' + key[i % key.size()] - 'a';
      cipherText += 'a' + (val % 26);
    }

    cout << "\nCipher Text: " << cipherText << endl;
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
}
else if (choice == 2)
{
  // Decryption
  string cipherText, key;


  cout << "\nEnter cipher text: ";
  getline(cin, cipherText);


  cout << "\nEnter key: ";
  getline(cin, key);


  // Removing spaces and converting to lowercase from key
  string temp2 = "";
  for (int i = 0; i < key.size(); i++)
  {
    if (key[i] != ' ')
      temp2 += key[i];
  }
  key = temp2;


  for (int i = 0; i < key.size(); i++)
  {
    if (key[i] >= 'A' && key[i] <= 'Z')
      key[i] += 32; // Convert to lowercase
  }


  // Decryption
  string decrypted = "";
  for (int i = 0; i < cipherText.size(); i++)
  {
    int val = cipherText[i] - 'a' - (key[i % key.size()] - 'a') + 26;
    decrypted += 'a' + (val % 26);
  }


  cout << "\nAfter decryption: " << decrypted << endl;
}
else
{
  cout << "Invalid choice. Please choose 1 or 2." << endl;
}
```

Name : Ritesh Pawar
PRN : 2020BTECS00068

```
  return 0;
}
```

Output:

Encryption:

Decryption:

Limitations:

Periodic Key Repeats: The Vigenère Cipher uses a keyword that repeats throughout the plaintext. The repetition of the keyword introduces patterns in the ciphertext, making it vulnerable to frequency analysis. If the keyword is short or has a recognizable pattern, cryptanalysts can deduce it, compromising the entire encryption.

Kasiski Examination: Cryptanalysts can use the Kasiski examination to identify repeated sequences in the ciphertext. By finding repeated sequences, they can estimate the length of the keyword and potentially recover parts of the key, making decryption easier.

Kerckhoffs's Principle Violation: The Vigenère Cipher violates Kerckhoffs's principle, which states that the security of a cryptographic system should not rely on the secrecy of the algorithm but only on the secrecy of the key. In the Vigenère Cipher, the security depends on both the key and the algorithm, which is a weakness.

Name : Ritesh Pawar
PRN : 2020BTECS00068

Key Length Determination: If the length of the keyword is unknown, determining the correct key length can be challenging. However, techniques such as the Friedman test and the Index of Coincidence can help cryptanalysts estimate the key length, making decryption more feasible.

Frequency Analysis Resistance is Limited: While the Vigenère Cipher is more resistant to simple frequency analysis compared to the Caesar Cipher, it is not entirely immune. Longer messages with repeating keywords can still exhibit frequency patterns that skilled cryptanalysts can exploit.