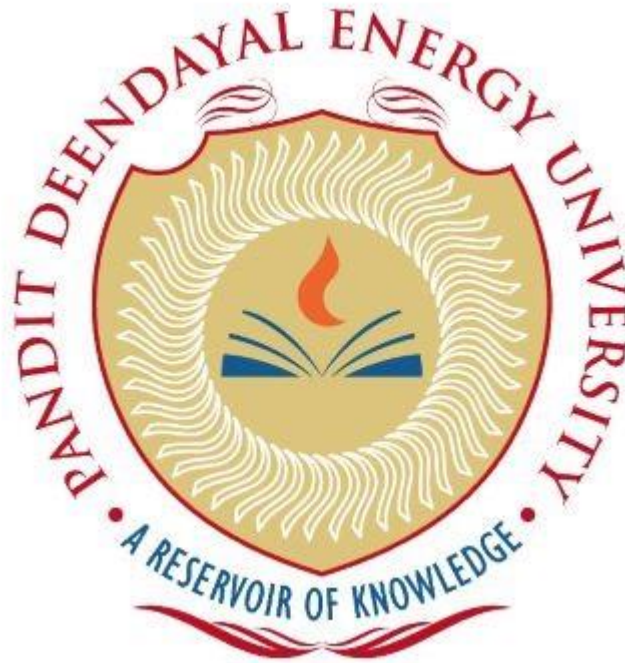


**PANDIT DEENDAYAL ENERGY UNIVERSITY
SCHOOL OF TECHNOLOGY**



**Course: Information Security Lab
Course Code: 20CP304P
LAB MANUAL
B.Tech. (Computer Science and Engineering)
Semester 5**

Submitted To:
Dr. Rutvij Jhaveri

Submitted By:
Harsh Shah
21BCP359
G11 batch

INDEX

S. No.	List of experiments	Sign
1	Download and practice Cryptool	
2	Study and Implement program for Caesar Cipher with Encryption, Decryption functions.	
3	Study and implement a program for Transposition Cipher to encrypt and decrypt a message	
4	Study and implement a program for Rail Fence Transposition Cipher to encrypt and decrypt a message	
5	Study and implement a program for Vigenère Cipher to encrypt and decrypt a message	
6	Study and implement a program for Playfair Cipher to encrypt and decrypt a message	
7	Study and implement a program for Hill Cipher to encrypt and decrypt a message	
8	Use Crypto++ library to implement encryption and decryption of different block ciphers	
9	Study and implement RSA encryption and decryption functions.	
10	Use RSA for generation and verification of digital signature on file	

Experiment No : 1

Aim : To Download and practice CrypTool.

Introduction : CrypTool is a program to Encrypt and Decrypt Text using different type of Ciphers. For this Lab Experiment we will use and try to get familiar with CrypTool

Output (CrypTool):

1. Caesar Cypher



Caesar / Rot13
Shifting cipher, which was used by Julius Caesar

Cipher | **Description**

Input (plaintext) length: 267

What is Lorem Ipsum?
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

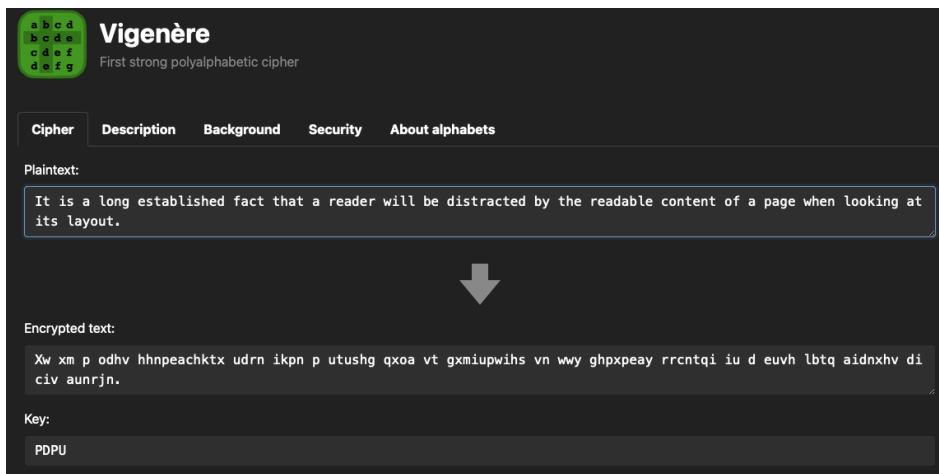
Encrypt ☒ Decrypt

Key: - 12 +

Output (ciphertext) length: 267

itmF uE XADqy UBEgy?
XADqy UBEgy uE EuyBxK pGyyK FqJF Ar Ftq BDuzFuzs mzp FKBqEqFFuzs uzpGEFDK. XADqy UBEgy tmE nqz Ftq uzpGEFDK'E
EFmzpmDp pGyyK FqJF qHqD EuZoq Ftq 1500E, Itqz mz GzWzAIz BDuzFqD FAAw m smxxqK Ar FKBq mzp EoDmynxqp uF FA ymwq
m FKBq EBqouyqz nAAw.

2. Vigenere Cipher



Vigenère
First strong polyalphabetic cipher

Cipher | **Description** | **Background** | **Security** | **About alphabets**

Plaintext:

It is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout.

↓

Encrypted text:

Xw xm p odhv hhnpeachktx udrn ikpn p utushg qxoa vt gxmiupwihs vn wvy ghpxpeay rrcntqi iu d euvh lbtq aidnxhv di civ aunrjn.

Key:

PDPu

3. Vernam Cipher

Ve

Vernam
Using XOR for implementing a one-time pad (OTP)

Cipher

Description

Plaintext:

Enter text ...

↓

Encrypted text:

WNFICDGKQAundefinedJDX

Key:

Same length key

4. Simple Columnar Transposition

Simple Columnar Transposition
Cipher that interchanges lines of the plaintext

Cipher

Description

Internal working

Input

Hello this is a test. This is a test text to test whether the text I entered is Correct.

length: 88

Keyword [according permutation: 1,4,5,3,2,6]

Cipher

length: 6

Encipher ☒ Decipher

Options

Alphabet

Show grid

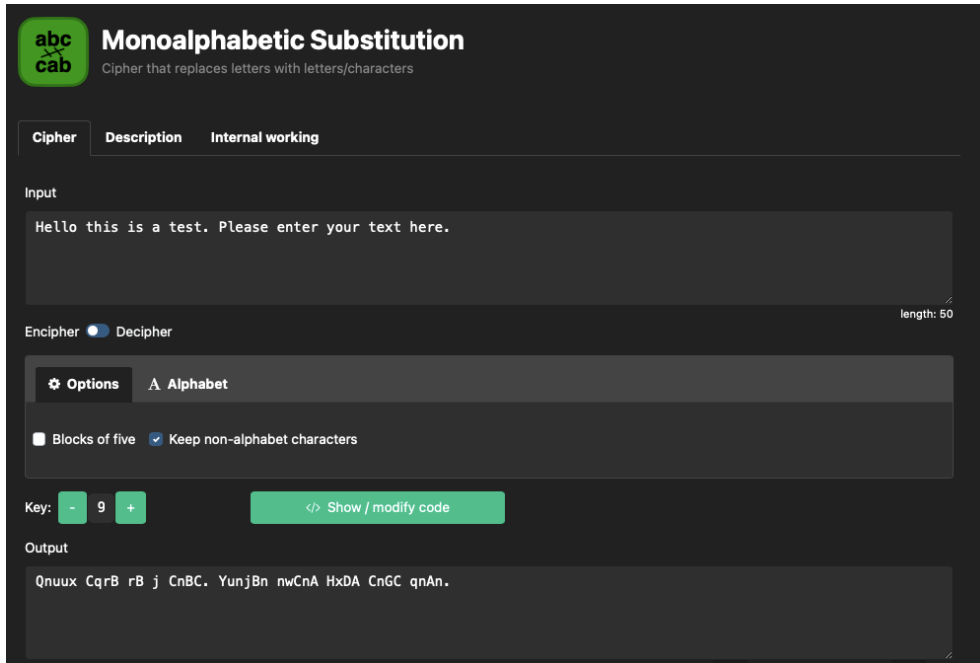
☐ Blocks of five ☐ Remove spaces ☒ Case-sensitive ☐ Replace spaces with

</> Show / modify code

Output [20 non-alphabet characters have been deleted]

HhtietteetscosTaeteedrlitsttehIereiesstwrxeCtllssitohttrotahtxshtnie

5. MonoAlphabetic Substitution



The screenshot shows the 'Monoalphabetic Substitution' tool interface. It has a dark theme with a green logo in the top left corner that says 'abc cab'. The title 'Monoalphabetic Substitution' is in white, with a subtitle 'Cipher that replaces letters with letters/characters' below it. There are three tabs: 'Cipher', 'Description', and 'Internal working', with 'Cipher' being the active tab. Below the tabs is an 'Input' section with a text area containing 'Hello this is a test. Please enter your text here.' and a 'length: 50' indicator. Below the input is a toggle for 'Encipher' (selected) and 'Decipher'. Below that is an 'Options' section with a dropdown menu set to 'A Alphabet'. There are two checkboxes: 'Blocks of five' (unchecked) and 'Keep non-alphabet characters' (checked). Below the options is a 'Key' section with a numeric keypad showing '9' and a 'Show / modify code' button. Below the key is an 'Output' section with a text area containing the encrypted text: 'Qnuux CqrB rB j CnBC. YunjBn nwCnA HxDA CnGC qnAn.'.

Cryptanalysis :

Applications :

References :

1. <https://www.cryptool.org/en/cto/caesar>
2. <https://www.cryptool.org/en/cto/vigenere>
3. <https://www.cryptool.org/en/cto/vernam>
4. <https://www.cryptool.org/en/cto/transposition>
5. <https://www.cryptool.org/en/cto/monoalpha>

Experiment No : 2

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

Introduction : The Caesar cipher is one of the simplest and oldest encryption techniques used to secure information. Named after Julius Caesar, who is said to have used this method to communicate secretly, the Caesar cipher is a type of substitution cipher where each letter in the plaintext is shifted a certain number of positions down or up the alphabet.

Here's how the Caesar cipher works:

Encryption:

To encrypt a message using the Caesar cipher, you choose a fixed number called the "key" or "shift." Each letter in the plaintext is replaced by a letter that is a fixed number of positions down the alphabet.

For example, with a shift of 3:

'A' becomes 'D'
'B' becomes 'E'
'C' becomes 'F'
...
'X' becomes 'A'
'Y' becomes 'B'
'Z' becomes 'C'

So, the plaintext "HELLO" would be encrypted as "KHOOR" with a shift of 3.

Decryption:

To decrypt the message, you reverse the process. You choose the same key but shift the letters in the opposite direction. In the case of a shift of 3, you would shift each letter 3 positions back up the alphabet.

So, the ciphertext "KHOOR" would be decrypted as "HELLO" with a shift of 3.

Program:

Encryption:

```
def encrypt(plaintext, k):  
    alphabet="ABCDEFGHIJKLMNOPQRSTUVWXYZ"  
  
    encrypted_text=""
```

```

for char in plaintext:
    if char.isalpha():
        upper=char.isupper()
        char_index=alphabet.index(char.upper())
        enc_index=(char_index+k)%26
        enc_char=alphabet[enc_index]
        if not upper:
            enc_char=enc_char.lower()
    else:
        enc_char=char

    encrypted_text += enc_char

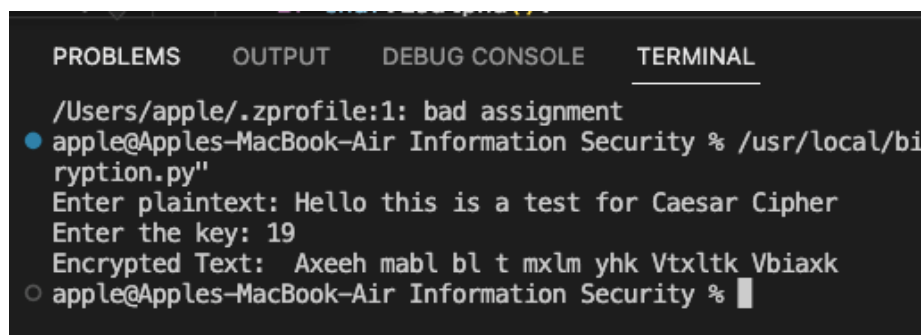
return encrypted_text

if __name__=="__main__":
    plaintext=input("Enter plaintext: ")
    key=int(input("Enter the key: "))

    encrypted_text=encrypt(plaintext,key)
    print("Encrypted Text: ", encrypted_text)

```

Output:



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
/Users/apple/.zprofile:1: bad assignment
● apple@Apples-MacBook-Air Information Security % /usr/local/bin/python3 encryption.py
Enter plaintext: Hello this is a test for Caesar Cipher
Enter the key: 19
Encrypted Text:  Axeeh mabl bl t mxlm yhk Vtxltk Vbiakx
○ apple@Apples-MacBook-Air Information Security %

```

Decryption:

```

def decrypt(cip_text, k):
    alphabet="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    dec_text=""

    for char in cip_text:
        if char.isalpha():
            upper=char.isupper()
            char_index=alphabet.index(char.upper())
            dec_index=(char_index-k)% 26
            dec_char=alphabet[dec_index]
            if not upper:
                dec_char=dec_char.lower()
        else:

```

```

        dec_char=char

        dec_text+=dec_char

    return dec_text

if __name__=="__main__":
    cip_text=input("Enter the ciphered text: ")
    key=int(input("Enter key: "))

    dec_text=decrypt(cip_text, key)
    print("this the decrypted text: ",dec_text)

```

Output:

```

/Users/apple/.zprofile:1: bad assignment
● apple@Apples-MacBook-Air Information Security % /usr/local/bin/python3
  ryption.py"
  Enter the ciphered text: Axeeh mabl bl t mxlm yhk Vtxltk Vbiaxk
  Enter key: 19
  this the decrypted text: Hello this is a test for Caesar Cipher
○ apple@Apples-MacBook-Air Information Security % █

```

CryptTool:

Encryption:

Select encryption type

Caesar Cipher (Substitution) ▾

Select position :

19

abcdefghijklmnopqrstuvwxyz
↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
tuvwxyzabcdefghijklmnopqrstuvwxyz

Hello this is a test for Caesar Cipher

Encrypt

Decrypt

Clear

Axeeh mabl bl t mxlm yhk Vtxltk Vbiaxk

Decryption:

Select encryption type: Caesar Cipher (Substitutio)

Select position: 19

abcdefghijklmnopqrstuvwxyz
 tuvwxyzabcdefghijklmnopqrs

Axeeh mabl bl t mxlm yhk Vtxltk Vbiaxk

Encrypt Decrypt Clear

Hello this is a test for Caesar Cipher

Copy

Cryptanalysis :

```

Enter the text: GzzgiqgzJgct
Decrypted Text: GzzgiqgzJgct
FyyfhpfyIfbs
ExxegoexHear
DwwdfndwGdzq
CvvcemcvFcyp
BuubdlbuEbzo
AttackatDawn
ZsszbjzsCzvm
YrryaiyrByul
XqqxzhxqAxtk
WppwygwpZwsj
VoovxfvoYvri
UnnuweunXuqh
TmmtvdtmWtpg
SllsucslVsof
RkkrtbrkUrne
QjjqsaqjTqmd
PiiprzpiSplc
OhhoqyohRokb
NggnpngxQnja
MffmowmfPmiz
LeelnvleOlhy
KddkmukdNkgx
JccjltjdMjfw
IbbiksibLiev
HaahjrhaKhdu
  
```

Cryptanalysis of the Caesar Cipher involves trying to decipher the encrypted message without knowing the shift value. Here are a few common techniques used to break the Caesar Cipher:

1. **Brute Force Attack:** Since there are only 25 possible shift values (excluding the trivial shift of 0), you can try all possible shifts and see which one produces a meaningful message. This is feasible due to the small key space, and automated tools can easily perform this task.
2. **Frequency Analysis:** This technique involves analysing the frequency of letters in the ciphertext. In English, some letters are used more frequently than others. By calculating the frequency distribution of letters in the ciphertext, you can compare it to the expected frequency distribution of English letters. The shift value that brings the frequencies closer to the expected distribution is likely the correct one.
3. **Known Plaintext Attack:** If you have access to a portion of the plaintext and its corresponding ciphertext, you can deduce the shift value by comparing the differences between the letters in both texts. This method is especially useful if the known plaintext includes common words or phrases.
4. **Dictionary Attack:** If you suspect that the original plaintext is in English, you can use a dictionary to help guess the correct shift. Apply all possible shifts to the ciphertext and check if any resulting text matches with words in the dictionary. This can narrow down the possibilities considerably.
5. **Pattern Recognition:** Sometimes, certain patterns or repeating groups of letters can be spotted in the ciphertext that give hints about the shift value. For example, if a repeated three-letter pattern appears, it could indicate a shift of 1 or 25.
6. **Cryptanalysis Tools:** There are online tools and software that can automate the process of breaking the Caesar Cipher. These tools usually employ a combination of the above techniques and can quickly find the correct shift value.

Applications:

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

- **Education:** It's a great tool to introduce beginners to the concepts of encryption and decryption in cryptography.
- **Puzzles and Challenges:** In recreational settings, it can add an element of mystery to puzzles, games, or coding challenges.

- Historical Context: It can be used in historical re-enactments to demonstrate ancient cryptographic techniques.
- Basic Obfuscation: For non-critical applications, it can offer simple data obfuscation.
- Introductory Coding: It can be used to teach basic programming skills involving strings and loops.
- Steganography: As a component in larger steganographic methods where information is hidden within other media.

References :

1. GeeksforGeeks
2. TutorialsPoint

Experiment No : 3

Aim : Study and Implement program for Simple Columnar Transposition Cipher with Encryption, Decryption functions.

Introduction : Simple columnar transposition is a basic encryption technique used in the field of cryptography. It falls under the category of transposition ciphers, which involve rearranging the letters of a message to obscure its content. Columnar transposition specifically involves writing the plaintext message in rows of a certain length and then reading it out column by column to create the ciphertext.

Program:

Encryption:

```
from math import ceil

def encrypt(plaintext, key):

    #create an empty dictionary and set the keys as character of key and values as
    group of plain texts read column wise
    dict_plaintext={}
    for char in key:
        dict_plaintext[char]=""

    for char_index_p, char_p in enumerate(plaintext):
        char=key[char_index_p % len(key)] #here for eg key=code and
        plaintext=myname then first it will take c=m,o=y,d=n,e=a,c=mm,d=ye...
        dict_plaintext[char]+=char_p

    #create a dictionary to store keys as indices and values as char of key
    dict={index: letter for index, letter in enumerate(key)}

    #sort the dictionary of key via the values
    sorted_dict={index: letter for index, letter in sorted(dict.items(), key=lambda
    item: (item[1], item[0]))}

    #now connect both the dictionaries having keys as indices and values as group
    of char of plaintext
    sorted_enc_dict={key1: dict_plaintext[value1] for key1, value1 in
    sorted_dict.items()}

    #print the encrypted text using the join func in the connected dict
    encrypt_text=''.join(value for value in sorted_enc_dict.values())
    print("The encrypted text is:" +encrypt_text)
```

```

if __name__=="__main__":
    plaintext=input("Enter plaintext:")
    key=input("Enter the key:")

    encrypt(plaintext,key)

```

Output:

```

/Users/apple/.zprofile:1: bad assignment
[6] 54990
/usr/local/bin/python3 "/Users/apple/Desktop/PDEU/Semester
● apple@Apples-MacBook-Air Information Security % /usr/local
.py"
Enter plaintext:Hello this is a test
Enter the key:PDEU
The encrypted text is:e s elt asHoistlhi t
○ apple@Apples-MacBook-Air Information Security % █

```

Decryption:

```

from math import ceil

def decrypt(encrypted_text, key):
    # Calculate the number of columns and rows
    no_cols = len(key)
    no_rows = ceil(len(encrypted_text) / no_cols)

    max_chars = len(encrypted_text) // len(key)

    # Create a dictionary to store keys as indices and values as characters of the
    key
    key_dict = {index: letter for index, letter in enumerate(key)}

    # Sort the dictionary of key via the values
    sorted_key_dict = {index: letter for index, letter in sorted(key_dict.items(),
key=lambda item: (item[1], item[0]))}

    # Create an empty dictionary to store the decrypted text
    decrypted_dict = {key1: "" for key1 in sorted_key_dict}

    # Calculate the remaining characters that need to be distributed
    remaining_chars = len(encrypted_text) % len(key)

    # Distribute the characters from the encrypted text to the appropriate columns
    col_index = 0

```

```
for col_letter in sorted_key_dict.values():
    num_chars = max_chars
    if col_index < remaining_chars:
        num_chars += 1
    decrypted_dict[col_letter] = encrypted_text[:num_chars]
    encrypted_text = encrypted_text[num_chars:]
    col_index += 1

# Construct the decrypted text by reading the matrix column by column
decrypted_text = ""
for row in range(no_rows):
    for col in key:
        decrypted_text += decrypted_dict[col][row]

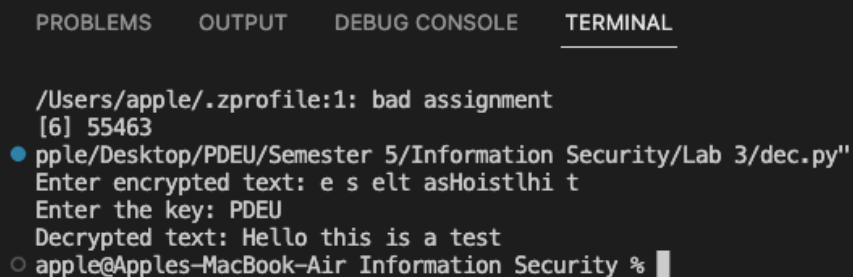
# Remove the underscore characters
decrypted_text = decrypted_text.replace("_", "")

return decrypted_text

if __name__ == "__main__":
    encrypted_text = input("Enter encrypted text: ")
    key = input("Enter the key: ")

    decrypted_text = decrypt(encrypted_text, key)
    print("Decrypted text:", decrypted_text)
```

Output:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

/Users/apple/.zprofile:1: bad assignment
[6] 55463
● pple/Desktop/PDEU/Semester 5/Information Security/Lab 3/dec.py"
Enter encrypted text: e s elt asHoistlhi t
Enter the key: PDEU
Decrypted text: Hello this is a test
○ apple@Apples-MacBook-Air Information Security %
```

CryptTool:**Encryption:**

Input

Hello this is a test

length: 20

Keyword [according permutation: 3,1,2,4]

PDEU

length: 4

Encipher ☒ Decipher

Options Alphabet Show grid

☐ Blocks of five ☐ Remove spaces ☒ Case-sensitive ☐ Replace spaces with

<> Show / modify code

Output [4 non-alphabet characters have been deleted]

etielhssHostliat

length: 16

Decryption:

Input

etielhssHostliat

length: 16

Keyword [according permutation: 3,1,2,4]

PDEU

length: 4

Encipher ☒ Decipher

Options Alphabet Show grid

☐ Blocks of five ☐ Remove spaces ☒ Case-sensitive ☐ Replace spaces with

<> Show / modify code

Output

Hellothisisatest

length: 16

Cryptanalysis :

Enumerate all Short Keywords

The first step in attacking a columnar transposition cipher is to try all possible short keywords. If we check all keywords up to a length of 9 or so, we don't have to wait very long. For every keyword permutation we score the deciphered text, then choose the text with the highest score as our best candidate. The number of possible rearrangements of a length N key is $N!$ (N factorial). This number grows very quickly as N gets larger. The number of possible keys for various length keywords is shown below

	Key Length	No. of permutations	Examples
22	AB, BA		
36	ABC, BAC, CBA, ...		
424	ABCD, ABDC, ACBD, ...		
5120	ABCDE, ABCED, ...		
6720	ABCDEF, ABDCFE, ...		
75,040	ABCDEFG, ABDCGEF, ...		
840,320	ABCDEFGH, ...		
9362,880	ABCDEFGHI, ...		

Trying to test all possible combinations of a length 6 keyword is easy, 720 trial decryptions can be done very quickly. However, we have little hope of trying to enumerate all possible length 12 keywords, as it would take far too long. This is why we stop at around length 9 keywords.

If this stage fails to decrypt the ciphertext, there are two possibilities: the actual key is longer than 9, or the key is shorter than 9 but the plaintext contains many strange quadgrams, which leads us to discount it as a possibility due to its low score. This can be overcome by ensuring the pre-generated quadgram statistics come from text as similar as possible to the ciphers we are breaking.

Dictionary Attacks :

If the first step failed, we now move on to the second. The columnar transposition cipher is almost always keyed with a word or short phrase, so we may not need to test all possible transposition keys, we may only need to test common words. This involves having a large list of dictionary words including place names, famous people, mythological names, historical names etc. From this we generate a text file of possible keys. We only need consider words longer than length 9, since we tested all the shorter words in step 1. We then try to decrypt the ciphertext with all possible dictionary words and record the keyword that resulted in plaintext with the highest quadgram fitness. Having 1,000,000 dictionary words would be a good comprehensive target.

This step can work if the keyword was one of the words that you included in the dictionary, but it fails to work if the keyword is something like 'THEBLOOMSINSEPTEMBER'. We can't hope to include all possible short phrases in our dictionary, as there are simply far too many of them. If this step fails, we must move on to step 3.

Hill Climbing:

The hill climbing approach we use here is almost identical to that used to break substitution ciphers. We first assume the key length is 10, then choose a random starting keyword of this length. This is called the parent key. Child keys are generated by making random swaps in the parent keyword, and if any of the swaps lead to an increased fitness we replace the parent with the child that beat it. In addition to randomly swapping two elements, we also try rotating the key e.g. 'ABCDEFGF' -> 'BCDEFGA', or cutting the key at a random point and swapping the ends e.g. 'ABCDEFGF' -> 'EFGABCD'. This allows us to better search the key space.

This algorithm will very rarely get the correct answer on the first run, typically you would have to run it several hundred times, each time starting with a different random key, to be sure to get the correct decryption. If after many tries the correct key is not found, it is time to increment the key length to 11 and rerun everything. This must be performed for key lengths from 10, 11, ... ,20. Keys of length 20 are very difficult to crack, and it gets much more difficult to crack keys as they become longer than 20. For long keys, length 20 and up, a simulated annealing algorithm would probably be a better choice and may reduce the number of iterations you need to perform to crack a cipher.

Applications :

Simple columnar transposition, despite its relatively low security level compared to modern encryption methods, still has some applications in specific scenarios:

- **Education and Learning:** Often used in educational settings to introduce students to the concept of encryption and basic cryptographic techniques. It provides a hands-on way to understand the principles of transposition ciphers and encryption processes.
- **Puzzles and Games:** Can be used to create puzzles and challenges for recreational purposes. Cryptogram puzzles, where a message is encrypted using a simple columnar transposition and the solver needs to decipher it, are a popular example.
- **Steganography:** Could be used as one of the techniques for hiding messages within larger texts or images.
- **Basic Data Obfuscation:** Might be used to obfuscate data temporarily or for fun. This could include hiding messages in public forums or within documents as a form of creative communication.
- **Historical Context:** Has historical significance as one of the early encryption methods used in wartime communications, particularly before the advent of modern cryptographic techniques. Studying its use can provide insights into the evolution of cryptography.

References :

3. [GeeksforGeeks](#)
4. [TutorialsPoint](#)

Experiment No : 04

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

Introduction :

The Rail Fence Cipher is a simple transposition cipher that rearranges the letters of a message to make it more difficult to decipher. It's called "Rail Fence" because when the letters are written in a zigzag pattern like railings of a fence. The cipher works by writing the message in a zigzag pattern across a set number of "rails" or lines, and then reading off the letters in a linear order.

Here's how the Rail Fence Cipher works:

- Choose the number of rails: Decide how many rails you want to use for the cipher. This number determines the height of the zigzag pattern.
- Write the message: Write the message you want to encrypt along the rails in a zigzag pattern. Start from the top rail and move diagonally downward until you reach the bottom rail, then reverse direction and move diagonally upward on the next rail.
- Read off the cipher text: Read the letters off in a linear order from the top rail to the bottom rail. This forms the encrypted message.

Program (Source Code):

```
# Program to implement RailFence Technique

def encrypt(text,key):
    text = text.upper().replace(' ','')

    # number of rows = key
    # number of columns = num of characters in text
    grid = [[" " for i in range(len(text))] for j in range(key)]

    # now two variables to declare row (for character to put in which row )and flag
    (for changing direction upwards and downwards)
    # if row = 0 then flag =0 (downwards)but if row = key-1 i.e last row then flag
    =1(upwards)
    row = 0
    flag =0

    for i in range(len(text)):
        grid[row][i]=text[i]
        if row == 0:
            flag=0
        elif row == key-1:
            flag = 1
        if flag ==0:
            row+=1
```

```
        else:
            row-=1

    for i in range(key):
        print("".join(grid[i]))

# now to read encrypted tag

encrypted_text=[]
for i in range(key):
    for j in range(len(text)):
        if grid[i][j] != ' ':
            encrypted_text.append(grid[i][j])
encrypted_text=''.join(encrypted_text)
return encrypted_text

def decrypt(cipher, key):

    grid = [['\n' for i in range(len(cipher))]
            for j in range(key)]
    flag = None
    row, col = 0, 0

    # mark the places where letters need to be placed with '*'
    for i in range(len(cipher)):
        if row == 0:
            flag = True
        if row == key - 1:
            flag = False #upper

        grid[row][col] = '*'
        col += 1

        if flag:
            row += 1
        else:
            row -= 1

    index = 0
    for i in range(key):
        for j in range(len(cipher)):
            if ((grid[i][j] == '*') and (index < len(cipher))):
                grid[i][j] = cipher[index]
                index += 1

    decrypted_text = []
    row, col = 0, 0
    for i in range(len(cipher)):

        # check the direction
```

```
        if row == 0:
            flag = True
        if row == key-1:
            flag = False

        if (grid[row][col] != '*'):
            decrypted_text.append(grid[row][col])
            col += 1

        # find the next row using
        # direction flag1
        if flag:
            row += 1
        else:
            row -= 1
    return("".join(decrypted_text))

while(True):
    print("1.Encrypt")
    print("2.Decrypt")
    print("3.Exit")
    choice =int(input("Enter the choice number:"))
    if choice ==1:
        text=input("Enrer the Plain Text: ")
        key =int(input("Enter the key: "))
        print("Encrypted Text: ",encrypt(text,key))
    elif choice ==2:
        text=input("Enter the Cipher Text: ")
        key =int(input("Enter the key: "))
        print("Decrypted Text: ",decrypt(text,key))
    elif choice == 3:
        print("Exit")
        break
    else:
        print("Invalid choice")
```

Output (Program):

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

/Users/apple/.zprofile:1: bad assignment
○ apple@Apples-MacBook-Air Information Security % /usr/local/b
1.Encrypt
2.Decrypt
3.Exit
Enter the choice number:1
Enrer the Plain Text: VANDEET SHAH
Enter the key: 3
V   E   H
A   D   E   S   A
N   T   H
Encrypted Text:  VEHADESANTH
1.Encrypt
2.Decrypt
3.Exit
Enter the choice number:2
Enter the Cipher Text: VEHADESANTH
Enter the key: 3
Decrypted Text:  VANDEETSHAH
1.Encrypt
2.Decrypt
3.Exit
Enter the choice number:█
```

Output (Cryptool):

Encryption:

Plaintext:

VANDEET SHAH

↓

Encrypted text:

VEHADES ANTH

Rendering:

V E H
A D E S A
N T H

Options:

Depth: Offset:

☐ filter whitespace characters ☐ group 5 characters ☐ filter non-alphabet characters ☐ convert to upper case

Decryption :

Plaintext:

VANDEETSHAH

↑

Encrypted text:

VEHADESANTH

Rendering:

```
V E H
A D E S A
N T H
```

Options:

Depth: 3 Offset: 0

☐ filter whitespace characters ☐ group 5 characters ☐ filter non-alphabet characters ☐ convert to upper case

Cryptanalysis :

1. **Brute Force Attack:** Since the Rail Fence Cipher has a limited number of possible keys (the number of rails), a brute force attack involves trying all possible rail numbers and deciphering the message. This can be done quickly, especially if the number of rails is small
2. **Pattern Recognition:** The Rail Fence Cipher leaves distinctive patterns in the ciphertext, making it susceptible to pattern recognition attacks. If the ciphertext is long enough, attackers can try to identify repeating sequences or other recognizable patterns.
3. **Known Plaintext Attack:** If the attacker has access to portions of both the plaintext and corresponding ciphertext, they can use this information to deduce the key and decrypt the entire message.

Applications :

1. The Rail Fence Cipher is simple to understand and implement, making it an ideal tool for introducing students to the concept of encryption and the principles behind transposition ciphers. It can serve as a hands-on example to teach basic cryptography concepts.
2. The Rail Fence Cipher can be used to create fun puzzles, challenges, or games for entertainment purposes. People who enjoy solving puzzles might find deciphering messages encrypted with the Rail Fence Cipher engaging.

References :

1. <https://www.tutorialspoint.com/>
2. <https://www.geeksforgeeks.org/>

Experiment No : 05

Aim : Study and implement a program for Vigenère Cipher to encrypt and decrypt a message.

Introduction :

The Vigenère Cipher is a polyalphabetic substitution cipher that uses a keyword to shift letters in the plaintext. Unlike the simple Caesar Cipher, the Vigenere Cipher employs multiple shifts in a repeating pattern based on the keyword. It is a more secure form of encryption.

Here's how the Vigenère Cipher works:

- Choose a keyword.
- Repeat the keyword to match the length of the plaintext.
- Encrypt the message by shifting each letter in the plaintext by the corresponding letter's position in the keyword.
- Decrypt the message by shifting the letters back in the opposite direction.

Program (Source Code):

```
def vigenere_encrypt(plain_text, key):
    encrypted_text = []
    key_length = len(key)
    for i in range(len(plain_text)):
        char = plain_text[i]
        if char.isalpha():
            key_char = key[i % key_length]
            shift = ord(key_char.upper()) - ord('A')
            if char.isupper():
                encrypted_char = chr(((ord(char) - ord('A') + shift) % 26) +
ord('A'))
            else:
                encrypted_char = chr(((ord(char) - ord('a') + shift) % 26) +
ord('a'))
            else:
                encrypted_char = char
            encrypted_text.append(encrypted_char)
    return ''.join(encrypted_text)

def vigenere_decrypt(encrypted_text, key):
    decrypted_text = []
    key_length = len(key)
    for i in range(len(encrypted_text)):
        char = encrypted_text[i]
        if char.isalpha():
            key_char = key[i % key_length]
            shift = ord(key_char.upper()) - ord('A')
            if char.isupper():
```



```
        decrypted_char = chr(((ord(char) - ord('A') - shift) % 26) +
ord('A'))
    else:
        decrypted_char = chr(((ord(char) - ord('a') - shift) % 26) +
ord('a'))
    else:
        decrypted_char = char
    decrypted_text.append(decrypted_char)
    return ''.join(decrypted_text)

while True:
    print("Vigenere Cipher Menu:")
    print("1. Encrypt")
    print("2. Decrypt")
    print("3. Quit")

    choice = input("Enter your choice (1/2/3): ")

    if choice == '1':
        plain_text = input("Enter the plain text: ")
        key = input("Enter the encryption key: ")
        encrypted_text = vigenere_encrypt(plain_text, key)
        print("Encrypted Text:", encrypted_text)

    elif choice == '2':
        encrypted_text = input("Enter the encrypted text: ")
        key = input("Enter the decryption key: ")
        decrypted_text = vigenere_decrypt(encrypted_text, key)
        print("Decrypted Text:", decrypted_text)

    elif choice == '3':
        print("Goodbye!")
        break

    else:
        print("Invalid choice. Please enter 1, 2, or 3.")

print()
```

Output (Program):

Encryption

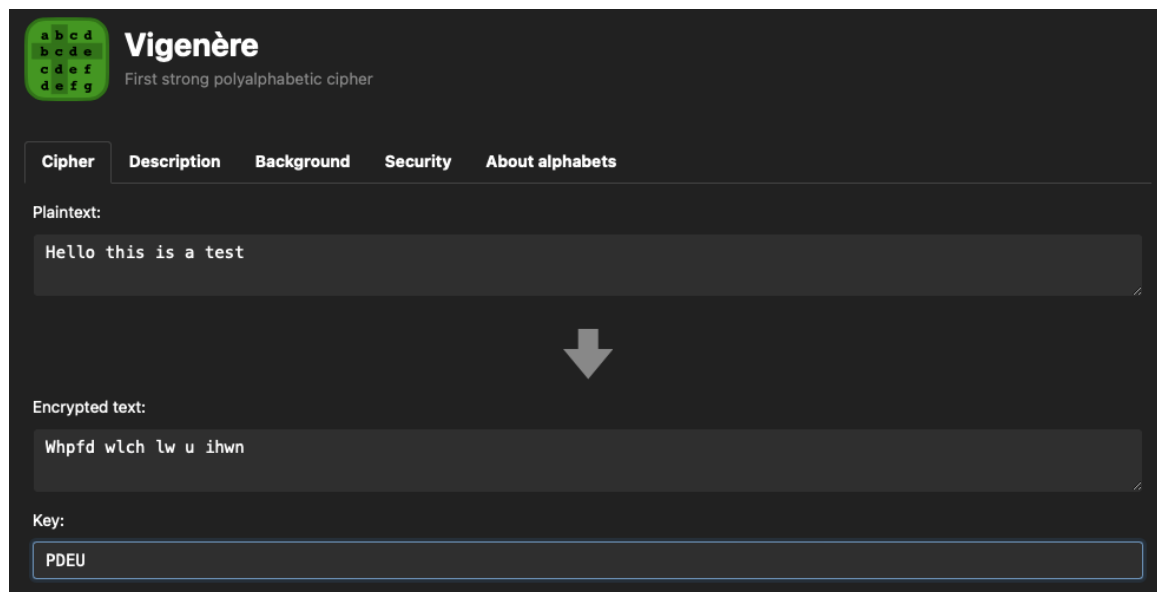
```
apple@Apples-MacBook-Air Information Security
Vigenere Cipher Menu:
1. Encrypt
2. Decrypt
3. Quit
Enter your choice (1/2/3): 1
Enter the plain text: Hello this is a test
Enter the encryption key: PDEU
Encrypted Text: Whpfd xbxv ch e ihwn
```

Decryption

```
apple@Apples-MacBook-Air Information Security
Vigenere Cipher Menu:
1. Encrypt
2. Decrypt
3. Quit
Enter your choice (1/2/3): 2
Enter the encrypted text: Lhpwdpindwlyyxaa
Enter the decryption key: PDEU
Decrypted Text: Welcometothejungle
```

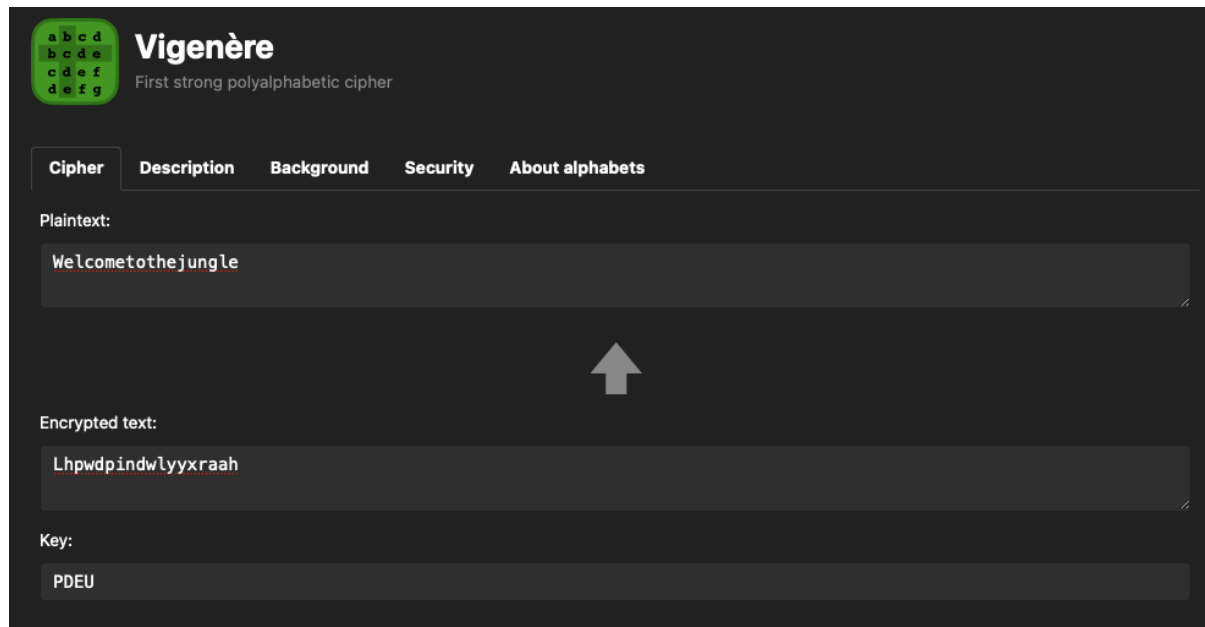
Output (Cryptool):

Encryption:



The screenshot shows the Cryptool Vigenère cipher interface. It features a title bar with a logo and the text 'Vigenère First strong polyalphabetic cipher'. Below the title bar are five tabs: 'Cipher', 'Description', 'Background', 'Security', and 'About alphabets'. The 'Cipher' tab is selected. The interface has three main input fields: 'Plaintext:', 'Encrypted text:', and 'Key:'. The 'Plaintext' field contains 'Hello this is a test'. The 'Encrypted text' field contains 'Whpfd wlch lw u ihwn'. The 'Key' field contains 'PDEU'. A large downward-pointing arrow is positioned between the 'Plaintext' and 'Encrypted text' fields, indicating the encryption process.

Decryption :



The screenshot shows a web-based Vigenère cipher tool. At the top left is a logo with a 3x3 grid of letters (a, b, c, d; b, c, d, e; c, d, e, f; d, e, f, g) and the text 'Vigenère' and 'First strong polyalphabetic cipher'. Below this is a navigation bar with tabs: 'Cipher', 'Description', 'Background', 'Security', and 'About alphabets'. The 'Cipher' tab is active. The interface has three main input fields: 'Plaintext:' containing 'Welcometothejungle', 'Encrypted text:' containing 'Lhpwdpindwlyyxr aah', and 'Key:' containing 'PDEU'. A large upward-pointing arrow is positioned between the plaintext and encrypted text fields, indicating the decryption direction.

Cryptanalysis :

4. **Brute Force Attack:** The Vigenère Cipher is susceptible to a brute force attack if the key length is known. An attacker can try all possible key combinations until the correct one is found.
5. **Frequency Analysis:** Like many classical ciphers, the Vigenère Cipher can be analysed using frequency analysis techniques if the key length is known. This involves looking for repeating patterns in the ciphertext.
6. **Kasiski Examination:** If there are repeated sequences in the ciphertext, Kasiski examination can be used to guess the key length and then perform a frequency analysis attack.

Applications :

1. The Vigenère Cipher has historical significance and was considered a secure encryption method in its time.
2. It can be used in educational settings to teach basic cryptography concepts and the importance of key management.
3. The Vigenère Cipher can be used for simple encryption in situations where strong security is not required.

References :

3. <https://www.tutorialspoint.com/>
4. <https://www.geeksforgeeks.org/>

Experiment No. - 6

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

Introduction:

The Playfair cipher, invented by Sir Charles Wheatstone and popularized by Lord Playfair, is a classical symmetric encryption method known for its use of a 5x5 letter matrix, the Playfair square. Operating on pairs of letters (bigrams) rather than individual characters, this polygraphic substitution cipher substitutes letter pairs to encrypt plaintext messages. While it has been surpassed by modern encryption techniques, the Playfair cipher offers historical insights into the evolution of cryptography.

Program:

Encryption:

```
def prepare_text(text):
    #remove spaces and convert to uppercase
    text=text.replace(" ", "").upper()
    #replace J with I
    text=text.replace("J", "I")

    valid_characters = "ABCDEFGHIIJKLMNOPQRSTUVWXYZ"

    # Filter out unwanted characters
    text = "".join(char for char in text if char in valid_characters)
    return text

def generate_key_matrix(key):
    #initialize the key matrix with all 0s
    matrix=[[' ' for _ in range(5)] for _ in range(5)]
    key=prepare_text(key)

    #fill the matrix with key and skip the duplicates
    key_chars=[]
    for char in key:
        if char not in key_chars:
            key_chars.append(char)

    alphabet="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

```
    row, col=0,0
    for char in key_chars:
        matrix[row][col]=char
        alphabet=alphabet.replace(char,"")
        col+=1
        if col==5:
            col=0
            row+=1

    for char in alphabet:
        matrix[row][col]=char
        col+=1
        if col==5:
            col=0
            row+=1

    return matrix

def find_positions(matrix,char):
    for row in range(5):
        for col in range(5):
            if matrix[row][col]==char:
                return row, col

def encrypt(plaintext, key):
    plaintext=prepare_text(plaintext)
    matrix=generate_key_matrix(key)

    enc_text=""

    for i in range(0, len(plaintext), 2):
        char1=plaintext[i]
        if i + 1 < len(plaintext):
            char2 = plaintext[i + 1]
        else:
            # If not, append a placeholder character or handle it as needed
            char2 = 'X'

        row1, col1 =find_positions(matrix, char1)
        row2, col2=find_positions(matrix, char2)

        if row1==row2:
            col1=(col1+1)%5
            col2=(col2+1)%5
        elif col1==col2:
            row1=(row1+1)%5
            row2=(row2+1)%5
```

```

        else:
            col1, col2=col2,col1

            enc_text+= matrix[row1][col1]+matrix[row2][col2]

    return enc_text

if __name__=="__main__":
    plaintext=input("Enter the plaintext: ")
    key=input("Enter the key: ")
    enc_text=encrypt(plaintext, key)
    print("Encrypted text: ", enc_text)

```

Output:

```

Enter the plaintext: Hello this is a test
Enter the key: Gemera
Encrypted text:  CANNUBOPLUMQRTU

```

Decryption:

```

def prepare_text(text):
    #remove spaces and convert to uppercase
    text=text.replace(" ", "").upper()
    #replace J with I
    text=text.replace("J", "I")

    valid_characters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    # Filter out unwanted characters
    text = "".join(char for char in text if char in valid_characters)
    return text

def generate_key_matrix(key):
    #initialize the key matrix with all 0s
    matrix=[['' for _ in range(5)] for _ in range(5)]
    key=prepare_text(key)

    #fill the matrix with key and skip the duplicates
    key_chars=[]
    for char in key:
        if char not in key_chars:
            key_chars.append(char)

```

```
alphabet="ABCDEFGHIJKLMNOPQRSTUVWXYZ"

row, col=0,0
for char in key_chars:
    matrix[row][col]=char
    alphabet=alphabet.replace(char,"")
    col+=1
    if col==5:
        col=0
        row+=1

for char in alphabet:
    matrix[row][col]=char
    col+=1
    if col==5:
        col=0
        row+=1

return matrix

def find_positions(matrix,char):
    for row in range(5):
        for col in range(5):
            if matrix[row][col]==char:
                return row, col

def encrypt(plaintext, key):
    plaintext=prepare_text(plaintext)
    matrix=generate_key_matrix(key)

    enc_text=""

    for i in range(0, len(plaintext), 2):
        char1=plaintext[i]
        if i + 1 < len(plaintext):
            char2 = plaintext[i + 1]
        else:
            # If not, append a placeholder character or handle it as needed
            char2 = 'X'

        row1, col1 =find_positions(matrix, char1)
        row2, col2=find_positions(matrix, char2)

        if row1==row2:
            col1=(col1+1)%5
            col2=(col2+1)%5
        elif col1==col2:
            row1=(row1+1)%5
```

```
        row2=(row2+1)%5
    else:
        col1, col2=col2,col1

    enc_text+= matrix[row1][col1]+matrix[row2][col2]

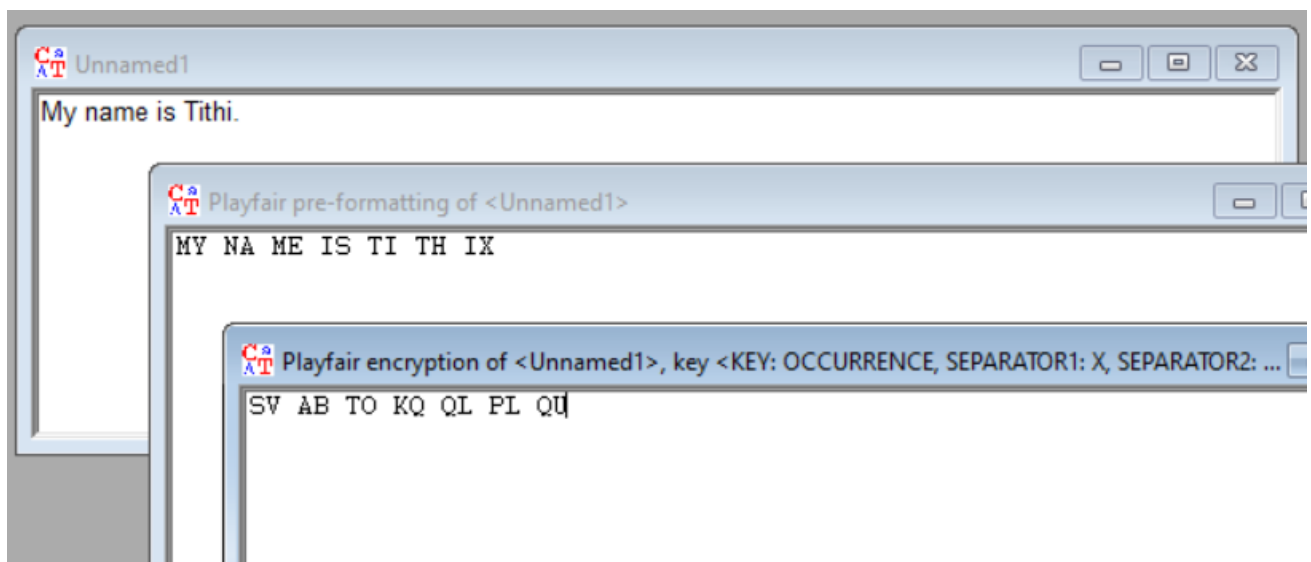
    return enc_text

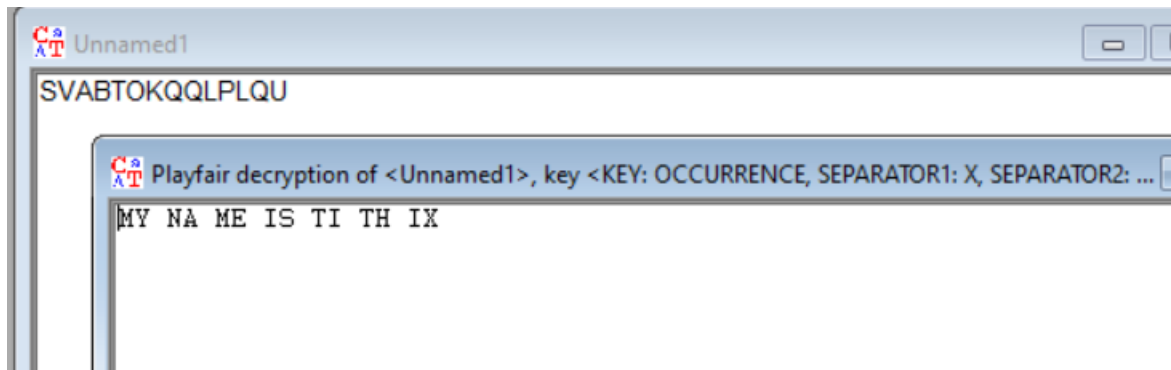
if __name__=="__main__":
    plaintext=input("Enter the plaintext: ")
    key=input("Enter the key: ")
    enc_text=encrypt(plaintext, key)
    print("Encrypted text: ", enc_text)
```

Output:

```
Enter the plaintext: CANNUBOPLUMQRTU
Enter the key: Gemera
Encrypted text: HE000THISISATEUP
```

Output (CrypTool):





Cryptanalysis:

The Playfair cipher is more complicated than a substitution cipher, but still easy to crack using automated approaches. It is known as a digraphic substitution cipher because pairs of letters are replaced by other pairs of letters. This obliterates any single letter frequency statistics, but the digraph statistics remain unchanged (frequencies of letter pairs). Unfortunately letter pairs have a much 'flatter' distribution than the single letter frequencies, so this complicates matters for solving the cipher using pen and paper methods.

Pen and paper methods of solving the playfair cipher are quite different to computer methods. The first step to a pen and paper method is usually to apply a 'crib', which is a known piece of plaintext to work out some of the key-square. This page will deal with solving plaintexts (around 100 characters or longer) with no crib using Simulated Annealing.

Applications:

The Playfair cipher, despite being an older and relatively simple encryption technique, has found applications in various domains, primarily historical and educational. Some of its applications include:

1. **Historical Encryption:**
The Playfair cipher was historically used for secure communication, especially during the late 19th and early 20th centuries when more advanced cryptographic methods were not widely available. It played a role in military and diplomatic communication during that era.
2. **Educational Tool:**
The Playfair cipher is often used as an educational tool to introduce students to the concepts of classical cryptography. It serves as a practical example for teaching encryption algorithms and techniques, helping students understand the fundamentals of encryption and decryption processes.

3. **Cryptography History:**
Studying the Playfair cipher is valuable for cryptography enthusiasts and historians. It offers insights into the evolution of cryptographic techniques, showcasing how encryption methods have evolved over time.
4. **Puzzle and Games:**
The Playfair cipher has also been employed in puzzles, brain teasers, and cryptographic games. Its use in recreational activities adds an element of challenge and historical charm.
5. **Cryptography Challenges:**
In some coding competitions and online cryptography challenges, the Playfair cipher might be included as a task for participants to solve. It provides an opportunity for enthusiasts to test their cryptographic skills.
6. **Personal Projects:**
Some individuals may choose to use the Playfair cipher for personal encryption needs or as part of hobbyist projects, especially if they have an interest in historical ciphers and cryptography.

It's important to note that while the Playfair cipher has historical significance and educational value, it is not considered a secure encryption method for modern communication due to its vulnerability to various cryptanalysis techniques. Modern cryptographic algorithms and techniques, such as those based on mathematical principles and strong key management, are used for secure data communication today.

References:

1. Practical Cryptography
2. GeeksforGeeks

Experiment No : 7

Aim: Study and implement a program for Hill Cipher.

Introduction:

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

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

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

Encryption:

To encrypt the text using hill cipher, we need to perform the following operation.

$$1. E(K, P) = (K * P) \bmod 26$$

Where **K** is the key matrix and **P** is plain text in **vector form**. Matrix multiplication of K and P generates the encrypted ciphertext.

Decryption:

To encrypt the text using hill cipher, we need to perform the following operation.

$$1. D(K, C) = (K^{-1} * C) \bmod 26$$

Where **K** is the key matrix and **C** is the ciphertext in **vector form**. Matrix multiplication of inverse of key matrix K and ciphertext C generates the decrypted plain text.

Program (Source Code):

```
def main():
    plaintext = "EXAM"
    key = "HILL"
    print(encrypt(plaintext, key))

def get_key_matrix(plaintext):
    cols = len(plaintext) // 2
    P = [[0] * cols for _ in range(2)]
    length = 0
    while length != len(plaintext):
        for i in range(2):
            for j in range(cols):
                P[i][j] = ord(plaintext[length]) - 65
                length += 1
    return P

def get_plain_text_matrix(key):
    K = [[0] * (len(key) // 2) for _ in range(2)]
    length = 0
    for i in range(len(key) // 2):
        for j in range(2):
            K[j][i] = ord(key[length]) - 65
            length += 1
    return K

def mat_mul(matrix1, r1, c1, matrix2, r2, c2):
    ans = [[0] * c2 for _ in range(r1)]
    for i in range(r1):
        for j in range(c2):
            ans[i][j] = 0
            for k in range(c1):
                ans[i][j] += matrix1[i][k] * matrix2[k][j]
    return ans

def encrypt(plaintext, key):
    length = len(plaintext)
    P = get_plain_text_matrix(plaintext)
    K = get_key_matrix(key)
    Encrypted = mat_mul(K, 2, len(key) // 2, P, 2, len(plaintext) // 2)

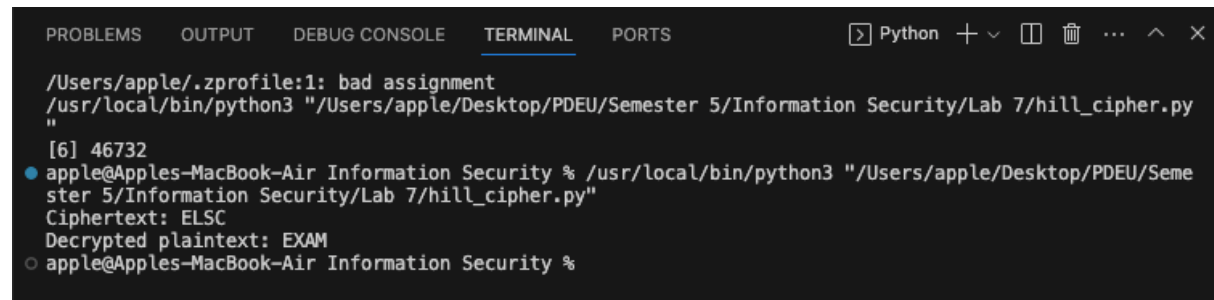
    encrypted = ""
    for i in range(len(plaintext) // 2):
        for j in range(2):
            encrypted += chr(Encrypted[j][i] % 26 + 65)

    return encrypted
```

```
if __name__ == "__main__":  
    main()
```

Note:-For Decryption we use same code which we use for encryption .In decryption for key we put inverse of KEY.

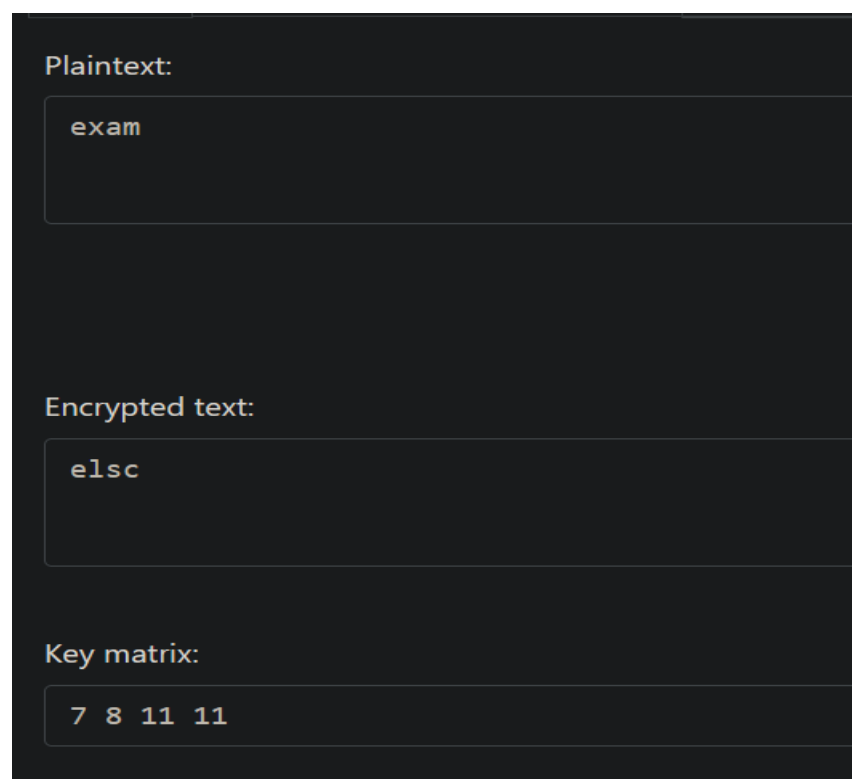
Output (Program):



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python + v [ ] [ ] ... ^ X  
/Users/apple/.zprofile:1: bad assignment  
/usr/local/bin/python3 "/Users/apple/Desktop/PDEU/Semester 5/Information Security/Lab 7/hill_cipher.py"  
[6] 46732  
apple@Apples-MacBook-Air Information Security % /usr/local/bin/python3 "/Users/apple/Desktop/PDEU/Semester 5/Information Security/Lab 7/hill_cipher.py"  
Ciphertext: ELSC  
Decrypted plaintext: EXAM  
apple@Apples-MacBook-Air Information Security %
```

Output (Cryptool):

Encryption Output (Cryptool):



Plaintext:

exam

Encrypted text:

elsc

Key matrix:

7 8 11 11

Decryption Output (Cryptool):

Cipher	Description	Background	Security
Plaintext:			
elsc			
Encrypted text:			
exam			
Key matrix:			
25 22 1 23			

Cryptanalysis :

Cryptanalysis involves analysing a cryptographic algorithm to uncover its weaknesses and vulnerabilities. While the Hill Cipher was historically considered more secure than simple substitution ciphers, it has certain vulnerabilities that can be exploited:

Key Length and Inverse: The security of the Hill Cipher relies on the key matrix being chosen correctly and kept secret. However, if an attacker can determine the key matrix (especially with small matrices), they can decrypt the ciphered text. In addition, some matrices might not have an inverse modulo 26, making decryption impossible.

Known-Plaintext Attack: If an attacker has access to pairs of plaintext and corresponding ciphertext, they can potentially recover the key matrix using mathematical methods. With enough pairs, they can solve a system of linear equations to find the key.

Brute Force Attack: If the attacker is unaware of the key matrix, they can attempt to brute-force the key matrix by trying all possible combinations. This becomes more computationally feasible with smaller key spaces.

Applications :

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

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

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

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

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

References:

<https://www.javatpoint.com/hill-cipher-program>

Experiment No : 8

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

Introduction:

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

Block Ciphers

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

The Crypto++ Library

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

AES Encryption and Decryption

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

DES and 3DES Encryption and Decryption

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

Program (Source Code):

```
#include <cryptopp/modes.h>
#include <cryptopp/aes.h>
#include <cryptopp/des.h>
#include <cryptopp/filters.h>
#include <cryptopp/hex.h>
#include <iostream>

using namespace CryptoPP;

int main() {
    std::string plaintext = "Hello, World!";
    std::string ciphertext;
    std::string decryptedtext;

    // AES
    {
        byte key[AES::DEFAULT_KEYLENGTH];
        memset(key, 0x00, AES::DEFAULT_KEYLENGTH);

        ECB_Mode< AES >::Encryption e;
        e.SetKey(key, AES::DEFAULT_KEYLENGTH);

        StringSource ss1(plaintext, true,
            new StreamTransformationFilter(e,
```

```
        new StringSink(ciphertext)
    )
};

ECB_Mode< AES >::Decryption d;
d.SetKey(key, AES::DEFAULT_KEYLENGTH);

StringSource ss2(ciphertext, true,
    new StreamTransformationFilter(d,
        new StringSink(decryptedtext)
    )
);

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

ciphertext.clear();
decryptedtext.clear();

// DES
{
    byte key[DES_EDE2::DEFAULT_KEYLENGTH];
    memset(key, 0x00, DES_EDE2::DEFAULT_KEYLENGTH);

    ECB_Mode< DES_EDE2 >::Encryption e;
    e.SetKey(key, DES_EDE2::DEFAULT_KEYLENGTH);

    StringSource ss1(plaintext, true,
        new StreamTransformationFilter(e,
            new StringSink(ciphertext)
        )
    );

    ECB_Mode< DES_EDE2 >::Decryption d;
    d.SetKey(key, DES_EDE2::DEFAULT_KEYLENGTH);

    StringSource ss2(ciphertext, true,
        new StreamTransformationFilter(d,
            new StringSink(decryptedtext)
        )
    );

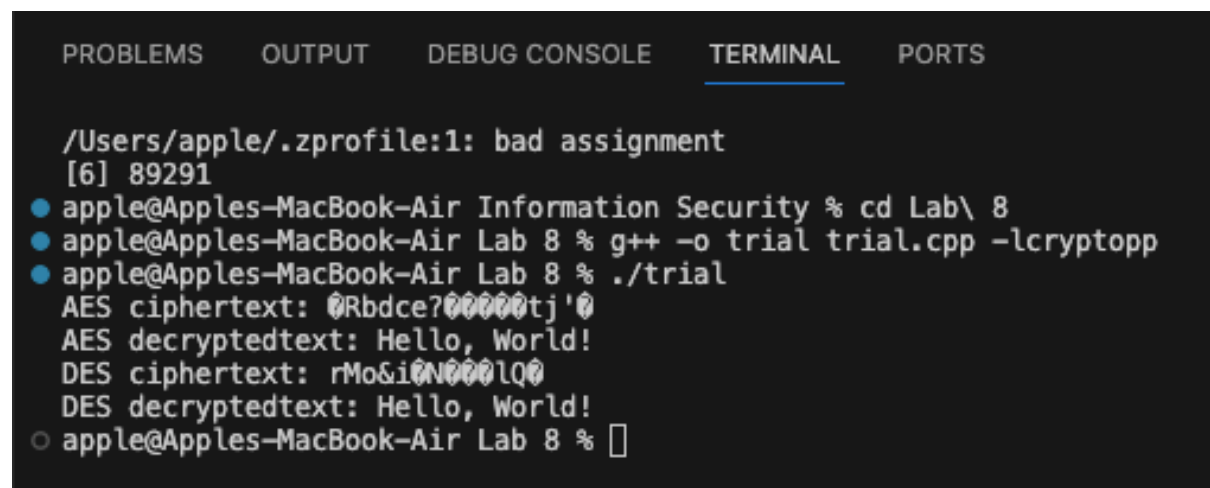
    std::cout << "DES ciphertext: " << ciphertext << std::endl;
    std::cout << "DES decryptedtext: " << decryptedtext << std::endl;
}

return 0;
}
```

How to Run the program?

1. Open the terminal
2. Navigate to the directory where you saved the file
3. Run the command `g++ -o trial {file name} -lcryptopp`
4. `./{file name}`

Output (Program):



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

/Users/apple/.zprofile:1: bad assignment
[6] 89291
● apple@Apples-MacBook-Air Information Security % cd Lab\ 8
● apple@Apples-MacBook-Air Lab 8 % g++ -o trial trial.cpp -lcryptopp
● apple@Apples-MacBook-Air Lab 8 % ./trial
AES ciphertext: 0Rbdce?000000tj'0
AES decryptedtext: Hello, World!
DES ciphertext: rMo&i0N000lQ0
DES decryptedtext: Hello, World!
○ apple@Apples-MacBook-Air Lab 8 %
```

Cryptanalysis :

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

1. **Brute Force Attacks:** One common form of cryptanalysis is a brute force attack, where an adversary tries all possible keys to decrypt a ciphertext. The security of a block cipher depends on the key length, as longer keys increase the computational effort required for a successful brute force attack.

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

Applications

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

1. **Data Encryption:** Block ciphers are used to encrypt sensitive data at rest or in transit, ensuring confidentiality and preventing unauthorized access. This includes encrypting files, databases, and email communications.
2. **Network Security:** Block ciphers are integral to secure network communication, such as SSL/TLS for securing web traffic, IPsec for VPNs, and SSH for secure remote access.

3. **Wireless Security:** Wireless protocols like WPA and WPA2 use block ciphers to protect Wi-Fi networks from eavesdropping and unauthorized access.
4. **Secure Messaging:** Messaging apps often employ block ciphers to encrypt text, voice, and video messages to safeguard user privacy.
5. **Secure Storage:** Block ciphers are used to encrypt data stored on devices, such as smartphones, ensuring that data remains confidential even if the device is lost or stolen.
6. **Financial Transactions:** Encryption of financial data is vital for secure online banking and payment processing, where block ciphers help protect sensitive information.

References:

<https://cryptopp.com>

Experiment No : 9

Aim: RSA encryption and decryption functions

Introduction:

RSA, which stands for Rivest–Shamir–Adleman, is a popular method for keeping our digital messages and information safe when they travel over the internet. Think of it as a special lock that only the right person can open.

Back in 1977, three smart folks named Ron Rivest, Adi Shamir, and Leonard Adleman, who worked at MIT, came up with RSA. They named it after their last names, and it's been a crucial tool for safeguarding emails and other online activities ever since.

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

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

This helps you: Keep your messages safe before sending them. And Ensure that the message hasn't been tampered with during its journey.

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

Program (Source Code):

```
import random
import math

def is_prime(n):
    if n < 2:
        return False
    for i in range(2, n // 2 + 1):
        if n % i == 0:
            return False
```

```
    return True

def generate_prime(min, max):
    prime = random.randint(min, max)
    while not is_prime(prime):
        prime = random.randint(min, max)
    return prime

def mod_inverse(e, phi):
    for d in range(3, phi):
        if (d*e)%phi == 1:
            return d
    raise ValueError("mod inverse does not exist")

p, q = generate_prime(1000, 5000), generate_prime(1000, 5000)

while p==q:
    q = generate_prime(1000, 5000)

n = p * q
phi_n = (p-1) * (q-1)

e = random.randint(3, phi_n-1)
while math.gcd(e, phi_n) != 1:
    e = random.randint(3, phi_n - 1)

d = mod_inverse(e, phi_n)

print("Public Key: ", e)
print("Private Key: ", d)
print("n: ", n)
print("Phi of n: ", phi_n)
print("p: ", p)
print("q:  ", q)

plaintext = "Hello World"

plaintext_encryption = [ord(ch) for ch in plaintext]
# (m ^ e)mod n = c
ciphertext = [pow(ch, e, n) for ch in plaintext_encryption]
ciphertext_alphabet = [chr((ch % 26) + 65) for ch in ciphertext]

print("Ciphertext: ", "".join(ciphertext_alphabet))

plaintext_decryption = [pow(ch, d, n) for ch in ciphertext]
decrypted_text = "".join(chr(ch) for ch in plaintext_decryption)
print("Plaintext after decryption: ", decrypted_text)
```

Output (Program):

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

/Users/apple/.zprofile:1: bad assignment
● apple@Apples-MacBook-Air Information Security % /usr/local
Public Key:  3723845
Private Key: 5886605
n: 6177659
Phi of n: 6172416
p: 3457
q: 1787
Ciphertext: USWWHYXHUWS
Plaintext after decryption: Hello World
○ apple@Apples-MacBook-Air Information Security %
```

Output (Cryptool):

Key generation

Encryption

Keys

Prime p

13

✓

Prime q

17

✓

coprime to:

☒ $\varphi(n)$ ☐ lcm

e

109

✓

Name

tushar

Generate random keypair

Alphabet [length: 256]:

☒ ASCII-256 ☐ Define your own alphabet

Separation:

Comma

▼

Encoding method:

b-adic

▼

Block length:

1

▼

Input type:

☐ Text ☒ Number

Numeral system:

☒ Decimal ☐ Octal ☐ Binary ☐ Hexadecimal

Input

☒ Manual ☐ File

13

Encryption into the ciphertext $c = m^e \bmod n$

13

Output from encryption in blocks of length 1

Cryptanalysis :

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

Ciphertext-Only Attacks: Imagine trying to read a coded message with no clues. RSA is pretty good at guarding against this because it uses tricky math.

Chosen-Plaintext Attacks: This is like knowing some parts of a secret message and trying to guess the rest. To protect against this, RSA uses a special "padding" to make it super hard for attackers to figure out the whole message.

Applications:

1. **Secure Communication**
2. **Digital Signatures**
3. **Secure Email**
4. **Secure File Transfer**

References:

<https://www.javatpoint.com/hill-cipher-program>

Experiment No : 10

Aim: RSA digital signature

Introduction:

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

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

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

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

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

Program (Source Code):

```
import random
import math

def is_prime(n):
    if n < 2:
        return False
    for i in range(2, n // 2 + 1):
        if n % i == 0:
            return False
    return True

def generate_prime(min, max):
    prime = random.randint(min, max)
    while not is_prime(prime):
        prime = random.randint(min, max)
    return prime

def mod_inverse(e, phi):
    for d in range(3, phi):
        if (d * e) % phi == 1:
            return d
    raise ValueError("Modular inverse does not exist")
```

```
# Function to sign a message with the sender's private key
def sign_message(message, d, n):
    message_bytes = message.encode()
    message_int = int.from_bytes(message_bytes, byteorder='big')
    signature = pow(message_int, d, n)
    return signature

# Function to verify the signature of a message using the sender's public key
def verify_signature(message, signature, e, n):
    message_bytes = message.encode()
    message_int = int.from_bytes(message_bytes, byteorder='big')
    decrypted_signature = pow(signature, e, n)
    return message_int == decrypted_signature

# Generate two prime numbers for the RSA key pair
p, q = generate_prime(1000, 5000), generate_prime(1000, 5000)

while p == q:
    q = generate_prime(1000, 5000)

n = p * q
phi_n = (p - 1) * (q - 1)

e = random.randint(3, phi_n - 1)
while math.gcd(e, phi_n) != 1:
    e = random.randint(3, phi_n - 1)

d = mod_inverse(e, phi_n)

print("Public Key (e):", e)
print("Private Key (d):", d)
print("n:", n)
print("Phi of n:", phi_n)
print("p:", p)
print("q:", q)

# Sender's Authentication
message_to_sign = "Sender is authentic"
signature = sign_message(message_to_sign, d, n)
print("Message to sign:", message_to_sign)
print("Signature:", signature)

is_authentic = verify_signature(message_to_sign, signature, e, n)
print("Is sender authentic?", is_authentic)

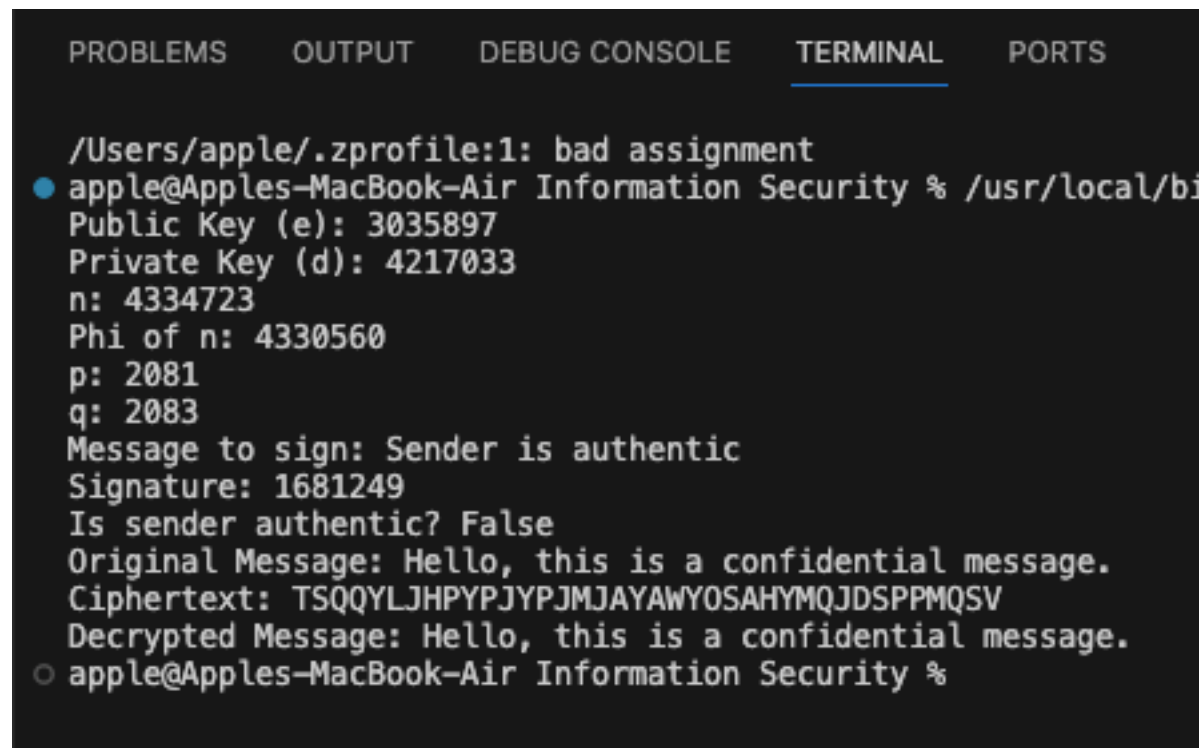
plaintext = "Hello, this is a confidential message."

plaintext_encryption = [ord(ch) for ch in plaintext]
ciphertext = [pow(ch, e, n) for ch in plaintext_encryption]
ciphertext_alphabet = [chr((ch % 26) + 65) for ch in ciphertext]
```

```
plaintext_decryption = [pow(ch, d, n) for ch in ciphertext]
decrypted_text = "".join(chr(ch) for ch in plaintext_decryption)

print("Original Message:", plaintext)
print("Ciphertext:", "".join(ciphertext_alphabet))
print("Decrypted Message:", decrypted_text)
```

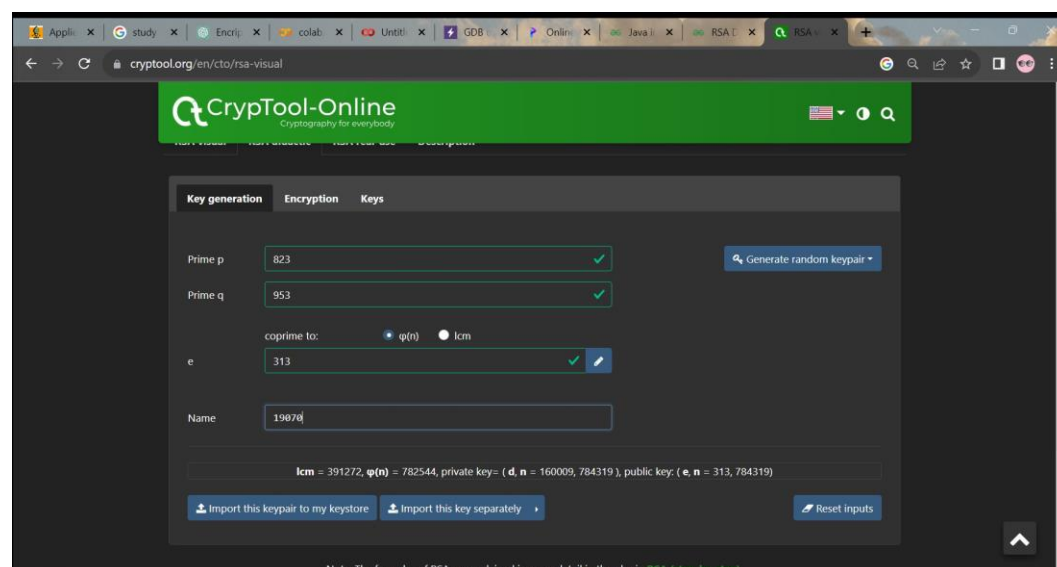
Output (Program):



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

/Users/apple/.zprofile:1: bad assignment
● apple@Apples-MacBook-Air Information Security % /usr/local/bi
Public Key (e): 3035897
Private Key (d): 4217033
n: 4334723
Phi of n: 4330560
p: 2081
q: 2083
Message to sign: Sender is authentic
Signature: 1681249
Is sender authentic? False
Original Message: Hello, this is a confidential message.
Ciphertext: TSQQYLJHPYPJYPJMJAYAWYOSAHYMQJDSPPMQSV
Decrypted Message: Hello, this is a confidential message.
○ apple@Apples-MacBook-Air Information Security %
```

Output (Cryptool):



CrypTool-Online
Cryptography for everybody

Key generation Encryption Keys

Prime p: 823 ✓

Prime q: 953 ✓

coprime to: ☒ $\phi(n)$ ☐ lcm

e: 313 ✓

Name: 19070

lcm = 391272, $\phi(n)$ = 782544, private key = (d, n = 160009, 784319), public key: (e, n = 313, 784319)

[Import this keypair to my keystore](#) [Import this key separately](#) [Reset inputs](#)

Note: The formulas of RSA are explained in more detail in the plugin [RSA \(step-by-step\)](#).

Applications:

1. **Secure Communication:** Digital signatures are used to ensure that messages or data exchanged between parties have not been tampered with during transmission. This is essential for secure email communication, instant messaging, and online chats.
2. **Software and Firmware Authentication:** Digital signatures are employed to verify the authenticity and integrity of software applications, updates, and firmware. This prevents the installation of malicious or tampered software.
3. **Document Authentication:** In legal and business contexts, digital signatures are used to verify the authenticity of electronic documents, contracts, and agreements. This eliminates the need for physical signatures and provides a secure method for electronic document management.
4. **E-commerce and Online Transactions:** Digital signatures play a crucial role in online payment systems and e-commerce. They ensure that online transactions are secure and that the transaction details are unaltered.

References:

<https://www.javatpoint.com/hill-cipher-program>