National Institute 0f Technology, Hamirpur

Department Of Computer Science and Engineering
July-Dec 2020

Subject Name: Information Security Lab	**Subject Code:** CSD – 415
Course: Information Security	**Semester:** 4^{th} Year, 7^{th} Semester
Submitted By: Student Name: Rishabh Katna Roll No: 17MI552	**Submitted To:** Dr. Narottam Chand Kaushal
	Faculty Signature:

<u>Experiment 1</u>

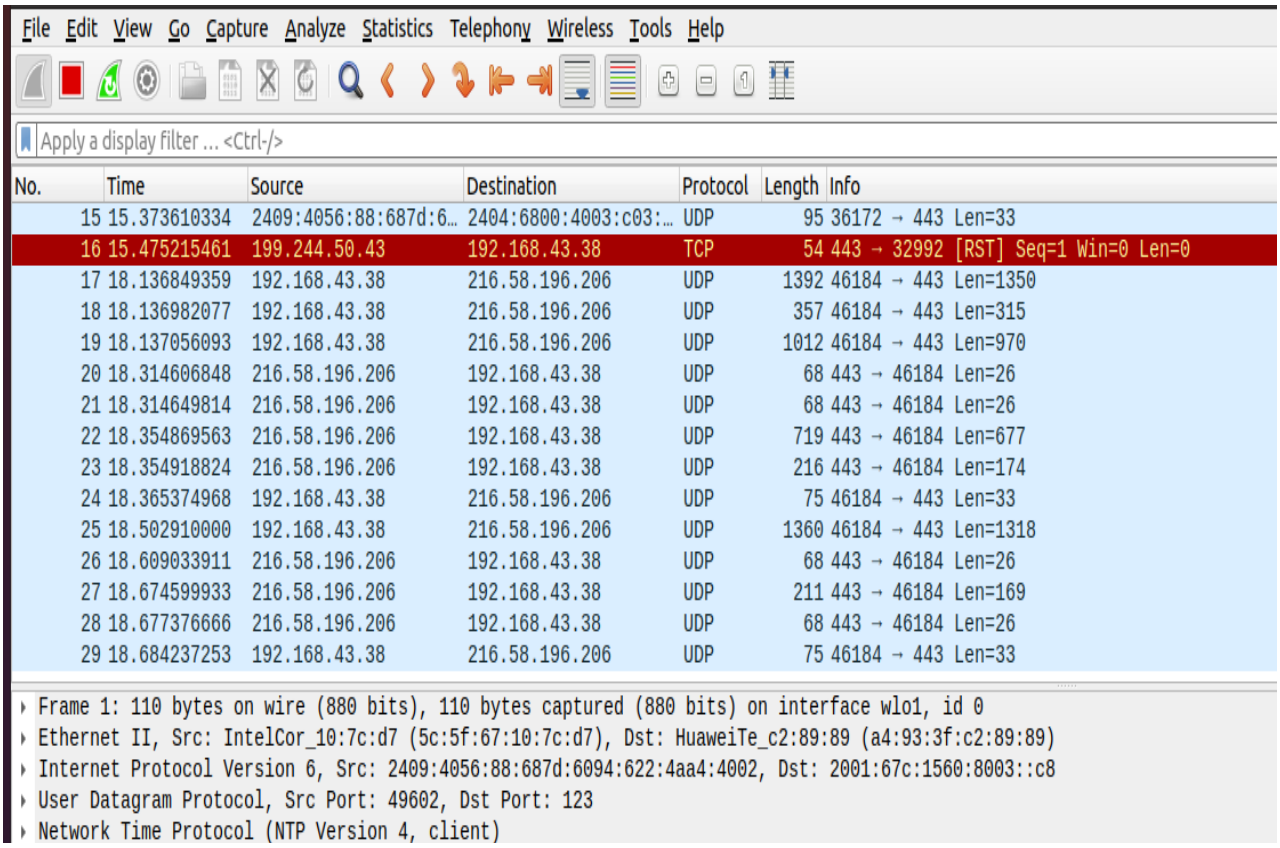
AIM : Introduction to Wireshark and implement Capture packets and Display packets in Wireshark.

Theory:

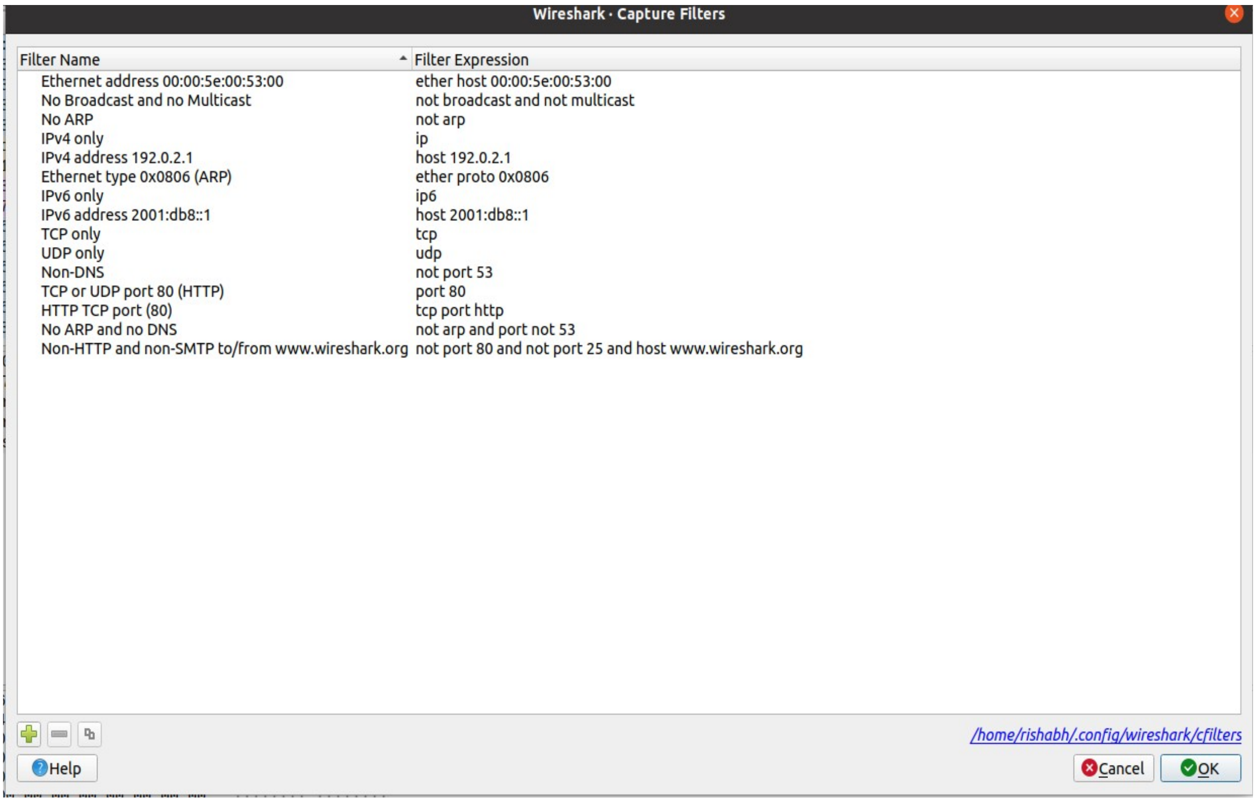
Wireshark, formerly known as Ethereal, can be used to examine the details of traffic at a variety of levels ranging from connection-level information to the bits that make up a single <u>packet</u>. Packet capture can provide a network administrator with information about individual packets such as transmit time, source, destination, <u>protocol</u> type and <u>header</u> data. This information can be useful for evaluating security events and troubleshooting network security device issues.

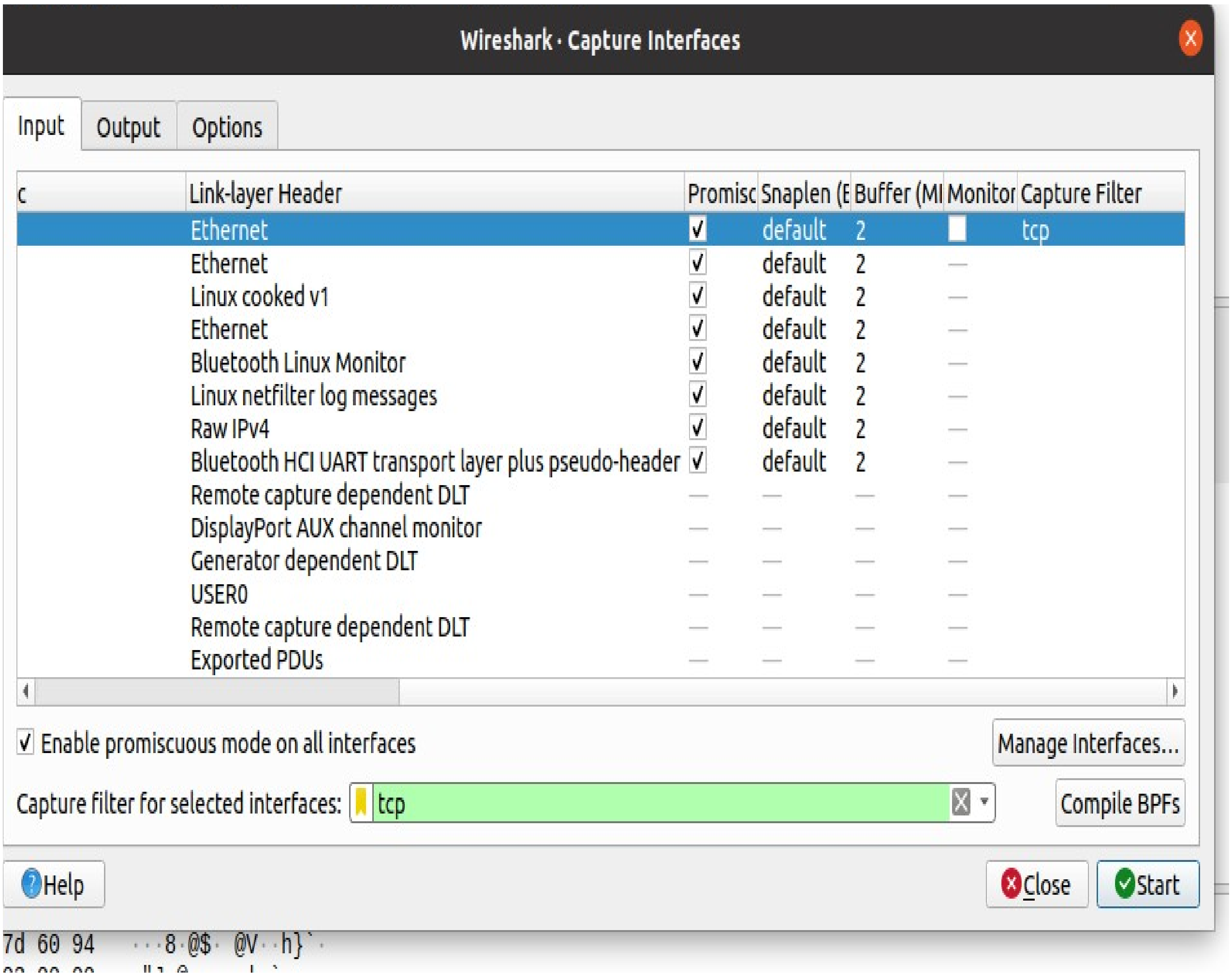
Images:

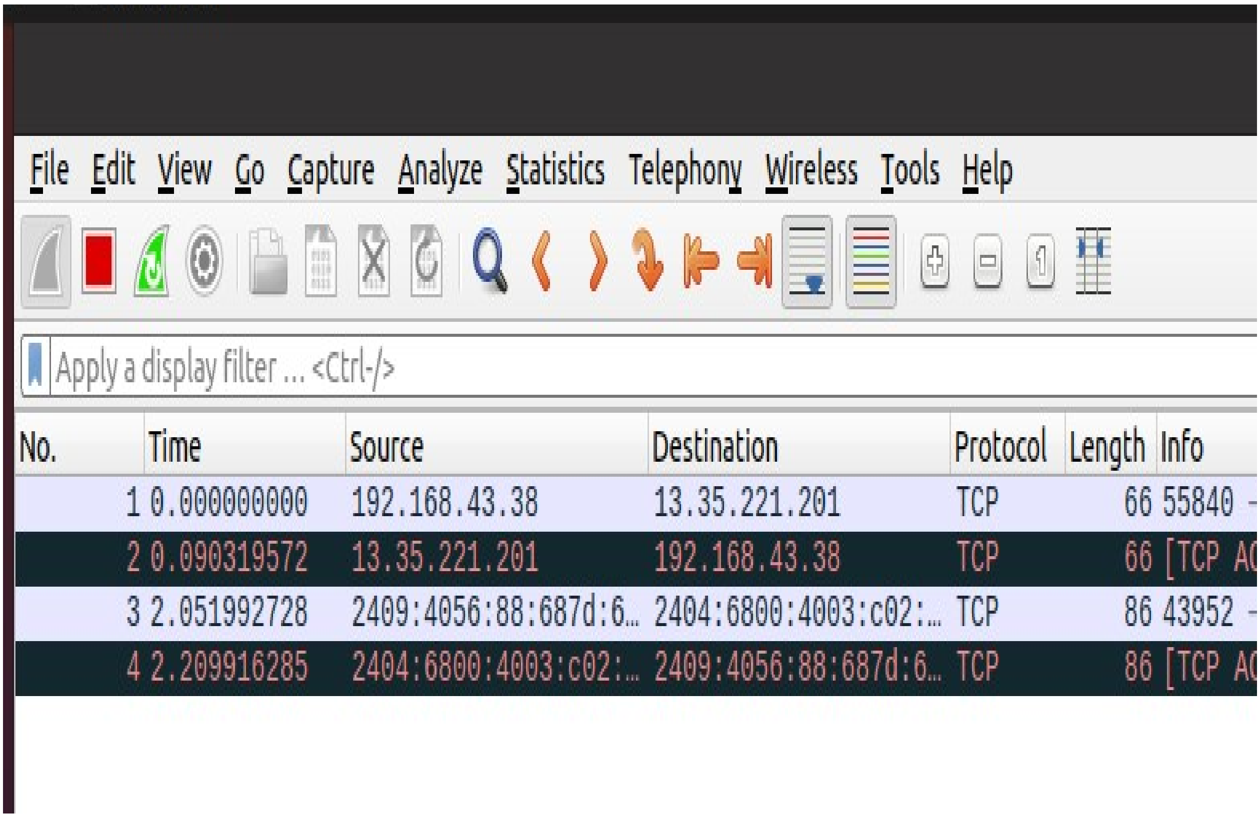
a. Interface of Wireshark:



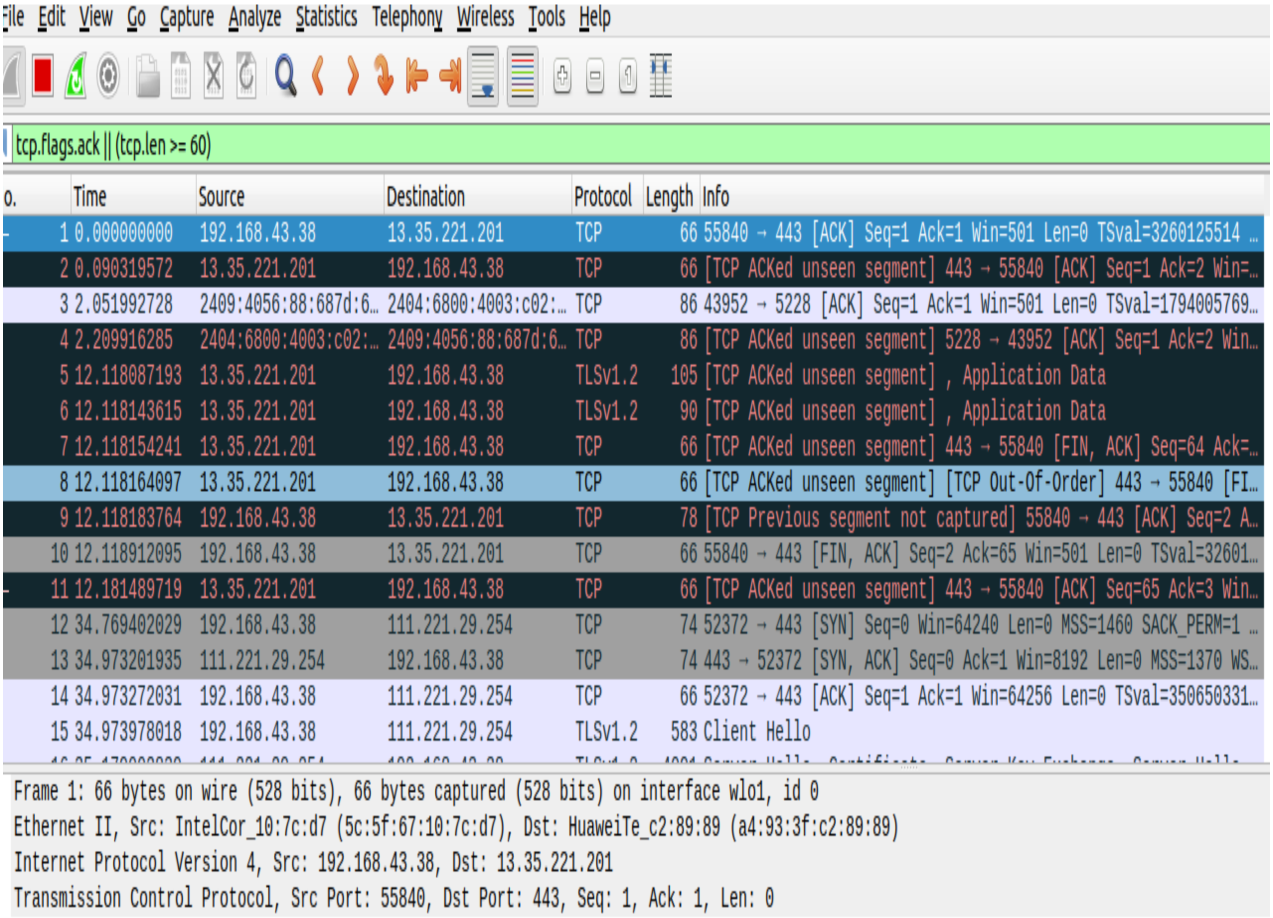
Capture packets filters in Wireshark:

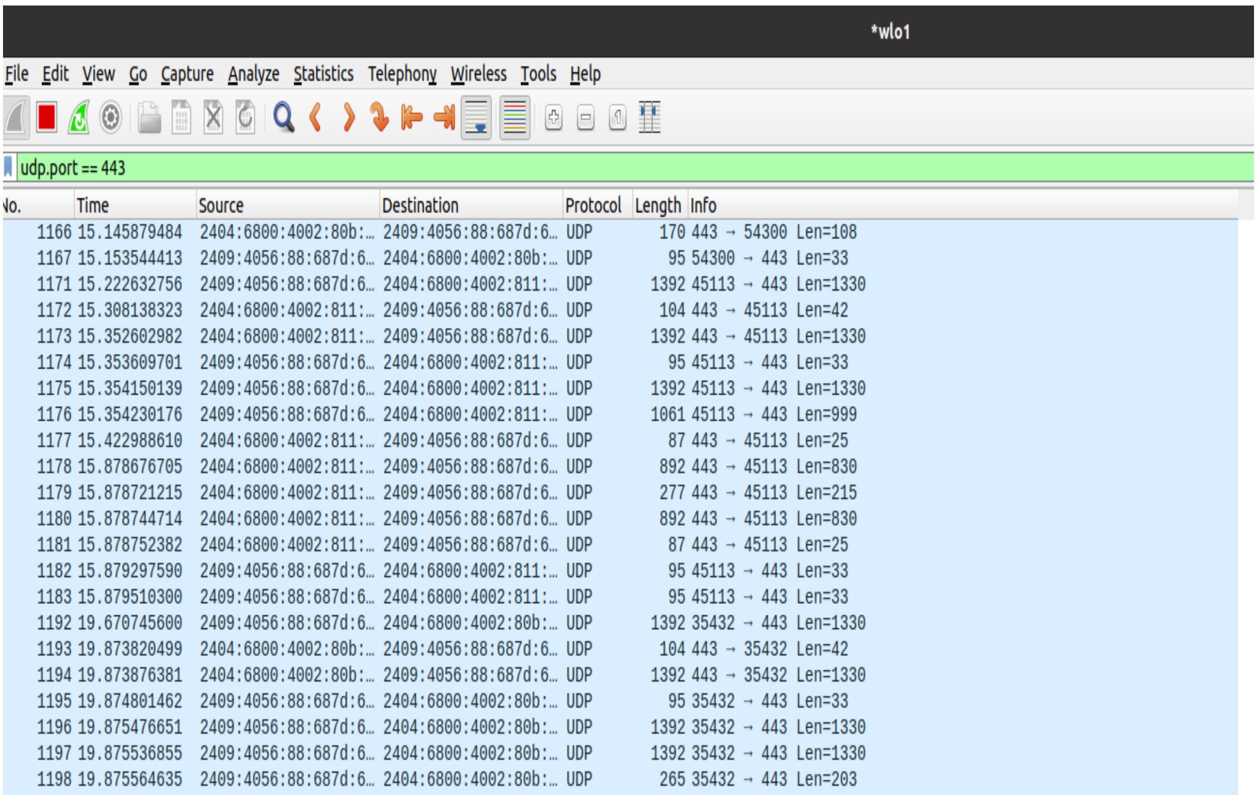



Wireshark · Capture Interfaces
Input
Output
Options
Link-layer Header
Promisc
Snaplen (B
Buffer (MB
Monitor
Capture Filter
Ethernet ✓ default 2 tcp
Ethernet ✓ default 2
Linux cooked v1 ✓ default 2
Ethernet ✓ default 2
Bluetooth Linux Monitor ✓ default 2
Linux netfilter log messages ✓ default 2
Raw IPv4 ✓ default 2
Bluetooth HCI UART transport layer plus pseudo-header ✓ default 2
Remote capture dependent DLT
DisplayPort AUX channel monitor
Generator dependent DLT
USER0
Remote capture dependent DLT
Exported PDUs
Enable promiscuous mode on all interfaces
Manage Interfaces...
Capture filter for selected interfaces:
tcp
Compile BPFs
Help
Close
Start
7d 60 94


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
No. Time Source Destination Protocol Length Info
1 0.000000000 192.168.43.38 13.35.221.201 TCP 66 55840
2 0.090319572 13.35.221.201 192.168.43.38 TCP 66 [TCP A
3 2.051992728 2409:4056:88:687d:6… 2404:6800:4003:c02:… TCP 86 43952
4 2.209916285 2404:6800:4003:c02:… 2409:4056:88:687d:6… TCP 86 [TCP A

Display Filter :





Experiment 2

Aim : To demonstrate the working of the following ciphers:

- **a. Caesar Cipher**
- **b. Hill Cipher**
- **c. Vigenere Cipher**

Theory:

a. Caesar Cipher:

The Caesar Cipher technique is one of the earliest and simplest method of encryption technique. It's simply a type of substitution cipher, i.e., each letter of a given text is replaced by a letter some fixed number of positions down the alphabet. For example with a shift of 1, A would be replaced by B, B would become C, and so on. The method is apparently named after Julius Caesar, who apparently used it to communicate with his officials.
Thus to cipher a given text we need an integer value, known as shift which indicates the number of position each letter of the text has been moved down.
The encryption can be represented using modular arithmetic by first transforming the letters into numbers, according to the scheme, A = 0, B = 1,…, Z = 25. Encryption of a letter by a shift n can be described mathematically as.

$$E_n(x) = (x + n) mod\ 26$$

(Encryption Phase with shift n)

$$D_n(x) = (x - n) mod\ 26$$

(Decryption Phase with shift n)

Code:

```
"""
Caesar Cipher Implementation python
"""
class CaesarCipher:
    def __init__(self, SHIFT):
        self.SHIFT = SHIFT


    # def is_capital(letter):

    def map_to_range(self, message):
        """
        Map a char string to characters having ascii (0, 25)
        """
        mapped_message = []
        for letter in message:
            if letter != ' ':
                map_value = ord(letter) - ord('a')
                # print(map_value)
                print(chr(map_value))
                mapped_message.append(chr(map_value))
            else:
                mapped_message.append(letter)

        return ''.join(mapped_message)



    def encrypt(self, message):
        """
        encrypts the message using caesar cipher

        => caesar cipher :
            in caesar cipher, a letter in a message is replaced with a SHIFT
places next to it.
        """
        mapped_message = self.map_to_range(message)

        encrypted_message = []

        for letter in mapped_message:
            if letter != " ":
                final_location = (ord(letter) + self.SHIFT) % 26
                encrypted_letter = chr(final_location)
                encrypted_message.append(encrypted_letter)
            else:
                encrypted_message.append(letter)

        return ''.join(encrypted_message)


    def decrypt(self, encoded_message):
        mapped_message = self.map_to_range(encoded_message)

        original_message = []

        for letter in mapped_message:
            if letter != " ":
                final_position = ord(letter) - self.SHIFT
                if final_position < 0:
                    final_position += 26
```

```
                original_message.append(chr(final_position))
            else:
                original_message.append(letter)

        return "".join(original_message)


def main():
    message = "this is a message"

    SHIFT = 4
    caesar_cipher = CaesarCipher(SHIFT)

    encrypted_text = caesar_cipher.encrypt(message)
    print(encrypted_text)

    original_message = caesar_cipher.decrypt(encrypted_text)
    print(original_message)

main()
```

b. Hill Cipher:

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

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

Code:

```
"""
Hill Cipher Python implementation
"""
import math
import numpy as np
import sympy

# plain_text = "attack"


class HillCipher:

    def __init__(self, key):
        self.key = key



################################################################################
###
    ########## helper functions
```

```
#######################################################
    def get_col_vectors(self, plain_text):
        """
        Convert text into column vectors
        e.g => plaintText = "abcd"
            => colVectors = [ ["a", "b"], ["c", "d"] ]
            => adjascent letters go into same column vector
        """
        list_column_vectors = []
        cur_col_vector = []

        len_cur_col_vector = 0

        for letter in plain_text:
            cur_col_vector.append(ord(letter) - ord('a'))
            len_cur_col_vector += 1
            if len_cur_col_vector == self.key.shape[0]:

                col_vector_as_array = np.array(cur_col_vector)
                # print(col_vector_as_array)

                list_column_vectors.append(col_vector_as_array)
                cur_col_vector.clear()
                len_cur_col_vector = 0
        return list_column_vectors

    def convert_col_vectors(self, list_column_vectors, key):
        """
        encrypt/decrypt the column vectors
        if encrypted => decrypt
        else encrypt
        """
        list_converted_col_vectors = []

        for column_vector in list_column_vectors:
            converted_col_vector = np.dot(key, column_vector) % 26
            # print(encrypted_col_vector)
            list_converted_col_vectors.append(converted_col_vector)

        return list_converted_col_vectors


    def get_converted_text(self, list_encrypted_col_vectors):
        """
        Convert the column vectors back to string form
        e.g => [["a", "b"], ["c", "d"]] ==> "abcd"
        """
        converted_text = []

        for col_vector in list_encrypted_col_vectors:
            first_char, second_char = col_vector
            converted_text.append(chr(math.floor(first_char) + ord('a')))
            converted_text.append(chr(math.floor(second_char) + ord('a')))


        converted_text = ''.join(converted_text)
        return converted_text



################################################################################
    ############### main encryption code
##########################################
```

```
    def encrypt(self, plain_text):
        """
        Encrypt the recieved plain text
        """
        list_column_vectors = []

        # key = np.array([[2, 3], [3, 6]])

        list_column_vectors = self.get_col_vectors(plain_text)


        list_encrypted_col_vectors = 
self.convert_col_vectors(list_column_vectors, self.key)


        encrypted_text = self.get_converted_text(list_encrypted_col_vectors)

        return encrypted_text


    ###########################################################################
    ###################### main decryption code ##############################
    def inverse(self, matrix):
        """
        compute inverse of a given matrix with mod(26)
        i.e => return inverse(matrix) mod(26)
        """
        determinant = int(np.linalg.det(matrix))

        determinant_inv = int(sympy.invert(determinant, 26))

        matrix_adjascent = np.linalg.inv(matrix) * determinant

        matrix_inv = determinant_inv * matrix_adjascent % 26

        matrix_inv = matrix_inv.astype(int)

        return matrix_inv


    def decrypt(self, encrypted_text):
        """
        decrypt the recieved encrypted text
        """
        key_inverse = self.inverse(self.key)

        # print(key_inverse)
        list_col_vectors = self.get_col_vectors(encrypted_text)

        list_decrypted_col_vectors = self.convert_col_vectors(list_col_vectors, 
key_inverse)

        original_text = self.get_converted_text(list_decrypted_col_vectors)

        # print(original_text)
        return original_text




def main():
    """
    Main function
```

```
    """
    key = np.array([[2, 3], [3, 6]])

    hill_cipher = HillCipher(key)

    plain_text = "plants"

    encrypted_text = hill_cipher.encrypt(plain_text)

    print(encrypted_text)

    original_text = hill_cipher.decrypt(encrypted_text)

    print(original_text)


main()
```

c.Vigenere Cipher:

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

•The table consists of the alphabets written out 26 times in different rows, each alphabet shifted cyclically to the left compared to the previous alphabet, corresponding to the 26 possible Caesar Ciphers.

•At different points in the encryption process, the cipher uses a different alphabet from one of the rows.

•The alphabet used at each point depends on a repeating keyword.

Code:

```
import numpy as np
def get_keystream(plain_text, key):
    """
    make the key as same size as plaintext by using repetitions
    eg:
    plainText = "thisismatch" ==> 11 letters
    key = "dog"
    keystream = "dogdogdogdo" ==> 11 letters
    """
    num_repetitions = int(len(plain_text) / len(key)) + 1

    key_stream_text = ''.join([key for _ in range(num_repetitions)])

    return key_stream_text[0:len(plain_text)]
```

```python
def encrypt(plain_text, key):
    """
    Encrypt the plain text using key
    """
    # get keystream from the key
    key_stream = get_keystream(plain_text, key)

    print(key_stream)


    vigenere_mat = []

    # convert adjascent pairs into vigenere matrix
    ## vigenere mat:
    ## eg: possible letters ==> "a", "b", "c"
    # vigenere mat -->
    # a b c
    # b c a   --> left shift(1 letter)
    # c a b   --> left shift(2 letters)
    for row in range(26):
        next_row = []
        for col in range(26):
            number = (row + col) % 26
            next_row.append(number)
        vigenere_mat.append(next_row)

    # print(vigenere_mat)
    encrypted_text = []

    for i in range(len(plain_text)):
        encrypted_char = vigenere_mat[ord(key_stream[i]) - ord('a')]
[ord(plain_text[i]) - ord('a')]
        encrypted_text.append(chr(encrypted_char + ord('a')))

    print("".join(encrypted_text))


    return plain_text



def main():
    plain_text = "attackatthedawn"
    key = "lemon"

    cipher_text = encrypt(plain_text, key)
    print(cipher_text)

main()
```

Output:

Caesar cipher output:

```
Original Message = this is a message xyztw

Encrypted Message = xlmw mw e qiwweki bcdxa

Decrypted Message = this is a message xyztw
```

Hill Cipher:

```
Orignal Message : plants
Encrypted Message : lhnaoj
Decrypted Message : plants
```

Vigenere Cipher:

```
$ python3 vignere_cipher.py
PlainText = attackatthedawn
Key = lemon
CipherText = lxfopvefhuphmka
rishabh@rishabh-HP-Pavilion-Lapto
```

Experiment 3

Aim : To demonstrate the working of playfair cipher

Theory:

The Playfair cipher was the first practical digraph substitution cipher. The scheme was invented in 1854 by Charles Wheatstone but was named after Lord Playfair who promoted the use of the cipher. In playfair cipher unlike traditional cipher we encrypt a pair of alphabets(digraphs) instead of a single alphabet.

Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_map>
#include <utility>

using namespace std;

#define Row vector<char>

class PlayfairCipher
{
    string key;
    vector<Row> Matrix;
    unordered_map<char, pair<int, int>> location;

    void createMatrix()
    {
        int rowIdx = 0, colIdx = 0;
        int i = 0;

        for (i = 0; i < key.size(); i++)
        {
            // else we are going to insert in sameLoc just using different
character
            if (location.find(key[i]) == location.end())
            {
                this->Matrix[rowIdx][colIdx] = key[i];

                // we didn't insert an entry in the hash table
                pair<int, int> newPair(rowIdx, colIdx);
                this->location[key[i]] = newPair;

                colIdx++;
                if (colIdx == 5)
                {
                    colIdx = 0, rowIdx++;
                }
            }
            // i++;
        }

        // fill the rest of the matrix
```

```cpp
        for (int i = 0; i < 26; i++)
        {
            char respectiveChar = i + 'a';
            if (respectiveChar == 'j')
                continue;

            if (location.find(respectiveChar) == location.end())
            {
                this->Matrix[rowIdx][colIdx] = respectiveChar;

                // we didn't insert an entry in the hash table
                pair<int, int> newPair(rowIdx, colIdx);
                this->location[respectiveChar] = newPair;

                colIdx++;
                if (colIdx == 5)
                {
                    colIdx = 0, rowIdx++;
                }
            }
        }
    }

    string improvePlainText(string plainText)
    {
        for (int i = 0; i < plainText.size(); i += 2)
        {
            if (plainText[i] == plainText[i + 1])
            {
                plainText.insert(plainText.begin() + i + 1, 'x');
            }
        }

        if (plainText.size() % 2 != 0)
            plainText += 'x';

        // cout << copyPlainText << ' ' << plainText << endl;

        return plainText;
    }

    void printMatrix()
    {
        for (Row r : Matrix)
        {
            for (char cell : r)
            {
                cout << cell << ' ';
            }
            cout << endl;
        }
    }

public:
    PlayfairCipher(string key)
    {
        this->key = key;

        this->Matrix.resize(5, Row(5));
        createMatrix();
        // printMatrix();
    }
```

```
    // ecnrypt function
    string encrypt(string plainText)
    {
        plainText = this->improvePlainText(plainText);
        // cout << plainText << endl;
        string cipherText = "";
        for (int i = 0; i < plainText.size() - 1; i += 2)
        {
            char firstChar = plainText[i], secondChar = plainText[i + 1];
            // cout << firstChar << ' ' << secondChar << endl;

            pair<int, int> locFirstChar = this->location[firstChar];
            pair<int, int> locSecondChar = this->location[secondChar];

            int rowFirstChar = locFirstChar.first, colFirstChar =
locFirstChar.second;
            int rowSecondChar = locSecondChar.first, colSecondChar =
locSecondChar.second;

            if (rowFirstChar == rowSecondChar)
            {
                cipherText += this->Matrix[rowFirstChar][(colFirstChar + 1) %
5];
                cipherText += this->Matrix[rowFirstChar][(colSecondChar + 1) %
5];
            }
            else if (colFirstChar == colSecondChar)
            {
                cipherText += this->Matrix[(rowFirstChar + 1) % 5]
[colFirstChar];
                cipherText += this->Matrix[(rowSecondChar + 1) % 5]
[colSecondChar];
            }
            else
            {
                cipherText += this->Matrix[rowFirstChar][colSecondChar];
                cipherText += this->Matrix[rowSecondChar][colFirstChar];
            }
        }

        // cout << cipherText << endl;
        return cipherText;
    }

    // decrypting function
    string decrypt(string cipherText)
    {
        string originalPlainText = "";
        for (int i = 0; i < cipherText.size() - 1; i += 2)
        {
            char firstChar = cipherText[i], secondChar = cipherText[i + 1];

            pair<int, int> locFirstChar = this->location[firstChar];
            pair<int, int> locSecondChar = this->location[secondChar];

            int rowFirstChar = locFirstChar.first, colFirstChar =
locFirstChar.second;
            int rowSecondChar = locSecondChar.first, colSecondChar =
locSecondChar.second;

            if (rowFirstChar == rowSecondChar)
            {
```

```
                int colOne = colFirstChar == 0 ? 4 : colFirstChar - 1;
                int colTwo = colSecondChar == 0 ? 4 : colSecondChar - 1;
                originalPlainText += this->Matrix[rowFirstChar][colOne];
                originalPlainText += this->Matrix[rowFirstChar][colTwo];
            }
            else if (colFirstChar == colSecondChar)
            {
                int rowOne = (rowFirstChar == 0) ? 4 : rowFirstChar - 1;
                int rowTwo = (rowSecondChar == 0) ? 4 : rowSecondChar - 1;
                originalPlainText += this->Matrix[rowOne][colFirstChar];
                originalPlainText += this->Matrix[rowTwo][colSecondChar];
            }
            else
            {
                originalPlainText += this->Matrix[rowFirstChar][colSecondChar];
                originalPlainText += this->Matrix[rowSecondChar][colFirstChar];
            }
        }

        return originalPlainText;
    }
};

int main()
{
    string plainText = "tattckatthedawn";
    cout << "Enter the text you want to encrypt : ";
    cin >> plainText;

    string key = "monarchy";

    PlayfairCipher pfCipher(key);

    string cipherText = pfCipher.encrypt(plainText);
    cout << "Text after encryption : " << cipherText << endl;

    string originalPlainText = pfCipher.decrypt(cipherText);
    cout << "Original text was : " << originalPlainText << endl;
}
```

Output:

```
playfair_cipher
Enter the text you want to encrypt : attackatthedawn
Text after encryption : rssrderspdkcnxaw
Original text was : attackatthedawnx
```

Experiment 4

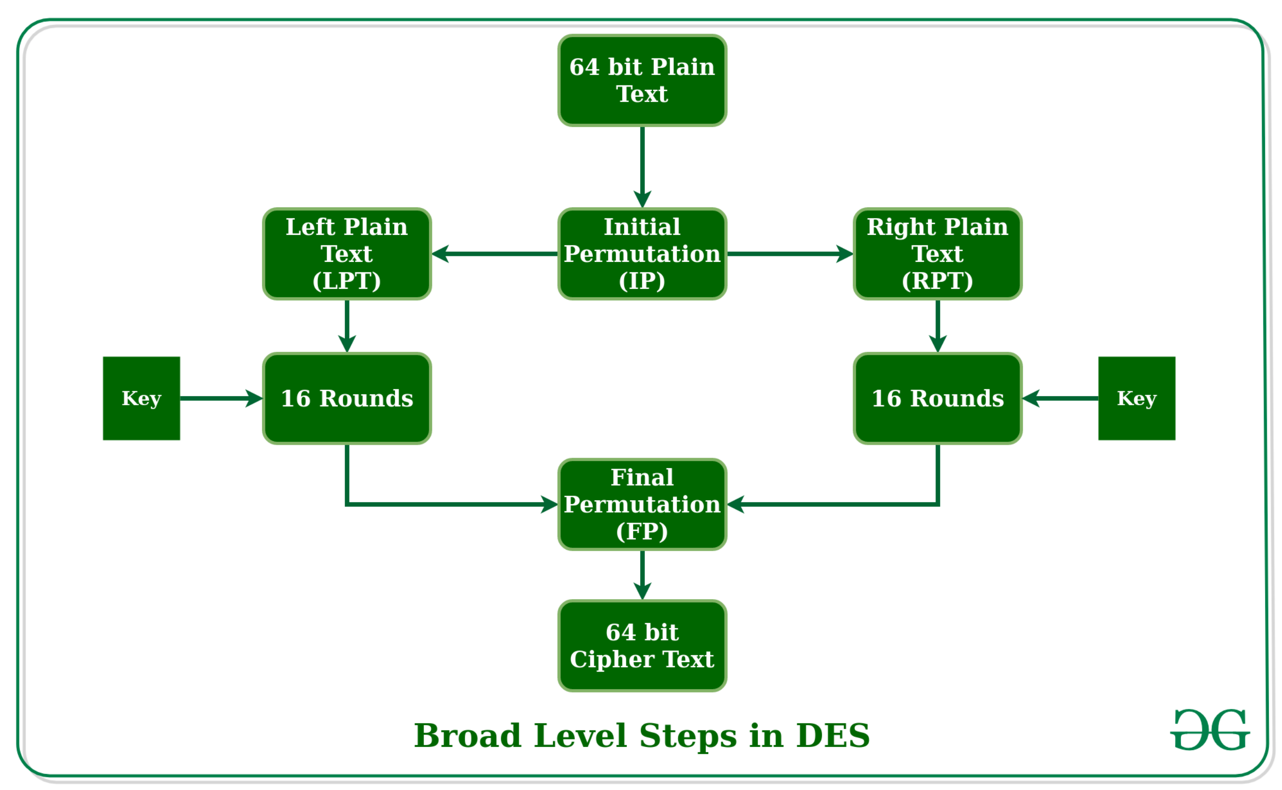
Aim : To demonstrate the working of DES(Data Encryption standard) and TripleDES algorithm.

Theory:

Data encryption standard (DES) has been found vulnerable against very powerful attacks and therefore, the popularity of DES has been found slightly on decline.

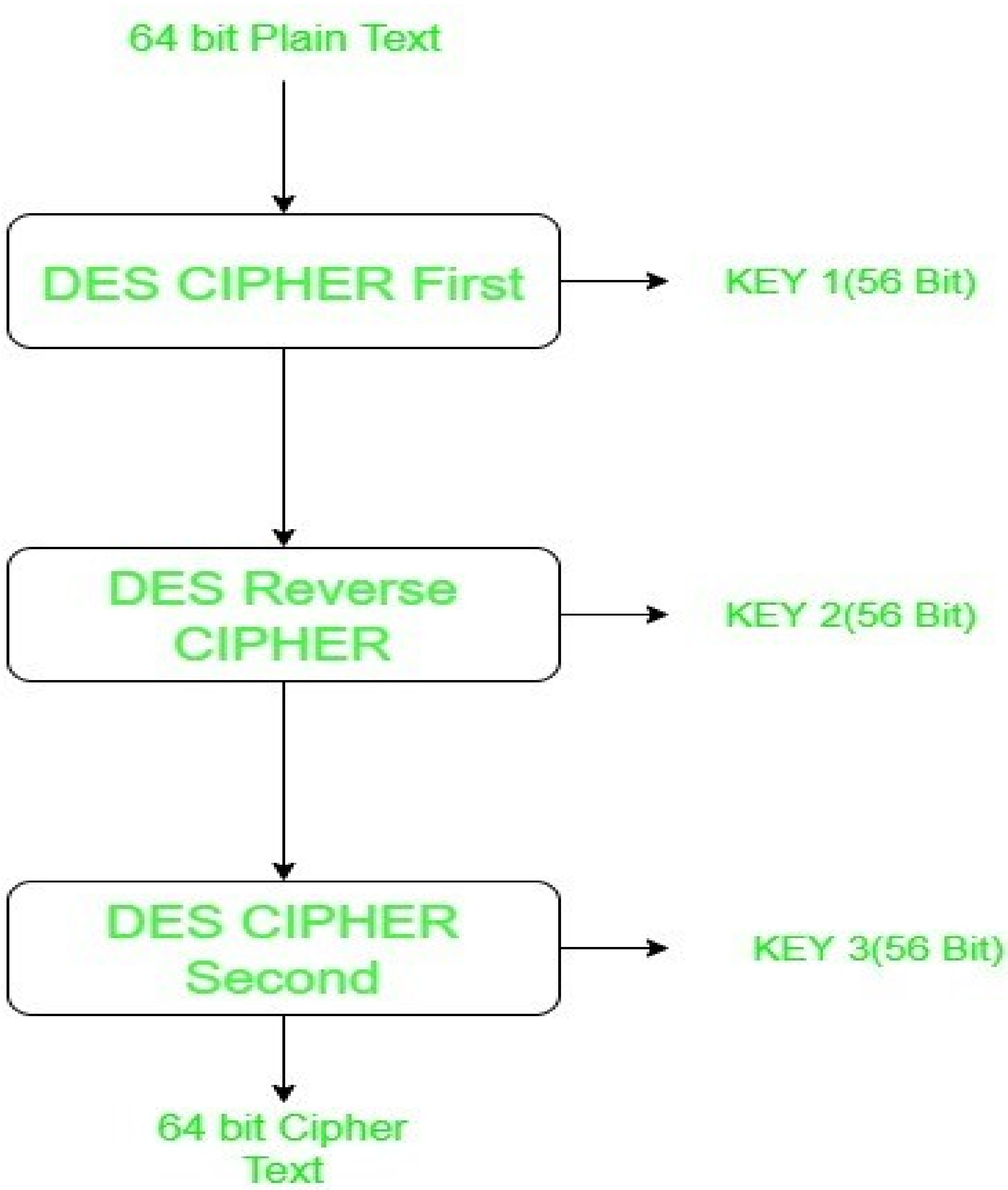
DES is a block cipher, and encrypts data in blocks of size of 64 bit each, means 64 bits of plain text goes as the input to DES, which produces 64 bits of cipher text. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is 56 bits.

STEPS IN DES:



Triple DES:

Triple DES is a encryption technique which uses three instance of DES on same plain text. It uses there different types of key choosing technique in first all used keys are different and in second two keys are same and one is different and in third all keys are same.



Code:

```
#include <iostream>
#include <climits>
#include <stdio.h>
#include <bitset>
#include <vector>
#include <algorithm>
#include <math.h>
#include <sstream>
#include <unordered_map>

using namespace std;
#define MOD_ENCRYPT 0
#define MOD_DECRYPT 1
int p_box_perm[32] = {16, 7, 20, 21,
                      29, 12, 28, 17,
                      1, 15, 23, 26,
                      5, 18, 31, 10,
                      2, 8, 24, 14,
                      32, 27, 3, 9,
                      19, 13, 30, 6,
                      22, 11, 4, 25};

int exp_perm[48] = {32, 1, 2, 3, 4, 5, 4, 5,
                    6, 7, 8, 9, 8, 9, 10, 11,
                    12, 13, 12, 13, 14, 15, 16, 17,
                    16, 17, 18, 19, 20, 21, 20, 21,
                    22, 23, 24, 25, 24, 25, 26, 27,
                    28, 29, 28, 29, 30, 31, 32, 1};

////////////////////////////////////////////////////////////////////////////////

// not working
string bin_to_hex(string s)
{
    // binary to hexadecimal conversion
    unordered_map<string, string> mp;
    mp["0000"] = "0";
    mp["0001"] = "1";
    mp["0010"] = "2";
    mp["0011"] = "3";
    mp["0100"] = "4";
    mp["0101"] = "5";
    mp["0110"] = "6";
    mp["0111"] = "7";
    mp["1000"] = "8";
    mp["1001"] = "9";
    mp["1010"] = "A";
    mp["1011"] = "B";
    mp["1100"] = "C";
    mp["1101"] = "D";
    mp["1110"] = "E";
    mp["1111"] = "F";
    string hex = "";
    for (int i = 0; i < (int)s.length(); i += 4)
    {
        string ch = "";
        ch += s[i];
        ch += s[i + 1];
        ch += s[i + 2];
        ch += s[i + 3];
        hex += mp[ch];
```

```
    }
    return hex;
}

string hex_to_bin(string hex_str)
{
    unordered_map<char, string> mp;
    mp['0'] = "0000";
    mp['1'] = "0001";
    mp['2'] = "0010";
    mp['3'] = "0011";
    mp['4'] = "0100";
    mp['5'] = "0101";
    mp['6'] = "0110";
    mp['7'] = "0111";
    mp['8'] = "1000";
    mp['9'] = "1001";
    mp['A'] = "1010";
    mp['B'] = "1011";
    mp['C'] = "1100";
    mp['D'] = "1101";
    mp['E'] = "1110";
    mp['F'] = "1111";
    string bin = "";
    for (int i = 0; i < (int)hex_str.size(); i++)
    {
        bin += mp[hex_str[i]];
    }
    return bin;
}

int s_box_perm[8][4][16] = {{14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7,
                             0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,
                             4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
                             15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13},
                            {15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10,
                             3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,
                             0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,
                             13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9},

                            {10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8,
                             13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
                             13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,
                             1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12},
                            {7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15,
                             13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
                             10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
                             3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2,
```

```
14},
                            {2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14,
9,
                             14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8,
6,
                             4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0,
14,
                             11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5,
3},
                            {12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5,
11,
                             10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3,
8,
                             9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11,
6,
                             4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8,
13},
                            {4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6,
1,
                             13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8,
6,
                             1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9,
2,
                             6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3,
12},
                            {13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12,
7,
                             1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9,
2,
                             7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5,
8,
                             2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6,
11}};

/*
    @params -> input string
    @return -> string after applying s-box permutation on it
    @Description -> It takes a 48 bit number, apply S-box perm and return a 32
bit output string
*/
string apply_s_box_perm(string input)
{

    string output = "";

    int n_groups = 8; // make 8 groups of 6 bits each
    for (int i_group = 0; i_group < n_groups; i_group++)
    {
        string cur_group_op = "";
        int start_idx = i_group * 6;

        string cur_str = input.substr(start_idx, 6);

        int s_table_row = (cur_str[0] - '0') * 2 + (cur_str[5] - '0');
        int s_table_col = bitset<4>(cur_str.substr(1, 4)).to_ulong();

        cur_group_op += bitset<4>(s_box_perm[i_group][s_table_row]
[s_table_col]).to_string();

        output += cur_group_op;

        // cout << i_group << " --> " << cur_str << " " << cur_group_op <<
endl;
```

```
    }

    return output;
}

/*
    @params -> permutation table(arr), input string, length of the output
string
    @return -> output string
    @Description -> output_string = permutation_table(input_string)
*/

int initial_perm[64] = {58, 50, 42, 34, 26, 18, 10, 2,
                        60, 52, 44, 36, 28, 20, 12, 4,
                        62, 54, 46, 38, 30, 22, 14, 6,
                        64, 56, 48, 40, 32, 24, 16, 8,
                        57, 49, 41, 33, 25, 17, 9, 1,
                        59, 51, 43, 35, 27, 19, 11, 3,
                        61, 53, 45, 37, 29, 21, 13, 5,
                        63, 55, 47, 39, 31, 23, 15, 7};
// encrypts data using this key

int pc2[48] = {14, 17, 11, 24, 1, 5,
               3, 28, 15, 6, 21, 10,
               23, 19, 12, 4, 26, 8,
               16, 7, 27, 20, 13, 2,
               41, 52, 31, 37, 47, 55,
               30, 40, 51, 45, 33, 48,
               44, 49, 39, 56, 34, 53,
               46, 42, 50, 36, 29, 32};

int pc1[56] = {57, 49, 41, 33, 25, 17, 9,
               1, 58, 50, 42, 34, 26, 18,
               10, 2, 59, 51, 43, 35, 27,
               19, 11, 3, 60, 52, 44, 36,
               63, 55, 47, 39, 31, 23, 15,
               7, 62, 54, 46, 38, 30, 22,
               14, 6, 61, 53, 45, 37, 29,
               21, 13, 5, 28, 20, 12, 4};

int final_perm[64] = {40, 8, 48, 16, 56, 24, 64, 32,
                      39, 7, 47, 15, 55, 23, 63, 31,
                      38, 6, 46, 14, 54, 22, 62, 30,
                      37, 5, 45, 13, 53, 21, 61, 29,
                      36, 4, 44, 12, 52, 20, 60, 28,
                      35, 3, 43, 11, 51, 19, 59, 27,
                      34, 2, 42, 10, 50, 18, 58, 26,
                      33, 1, 41, 9, 49, 17, 57, 25};
string apply_perm(int *arr, string input, size_t output_len)
{
    string output;

    output.resize(output_len);

    for (size_t i = 0; i < output_len; i++)
    {
        output[i] = input[arr[i] - 1];
    }

    return output;
}

int l_shift_table[16] = {1, 1, 2, 2,
```

```
                          2, 2, 2, 2,
                          1, 2, 2, 2,
                          2, 2, 2, 1};

string L_shift(string input, int num_steps)
{

    string append_end = input.substr(0, num_steps);

    // remove
    input.erase(input.begin(), input.begin() + num_steps);

    input += append_end;

    return input;
}

/**********************************************
 *
 * @params -> 64 bit key
 * @return -> list of 16 keys(K1 -> k16) to be used for each round
 *
 * *********************************************/
vector<string> prepare_compressed_keys(const string &key)
{
    // apply pc1
    vector<string> compressed_keys(16);

    /* Convert 64 bit key to 56 bit key */
    string key_final = apply_perm(pc1, key, 56);

    string l_key = key_final.substr(0, 28); // left half

    string r_key = key_final.substr(28, 28); // right half

    int n_rounds = 16;

    for (int i_round = 0; i_round < n_rounds; i_round++)
    {
        l_key = L_shift(l_key, l_shift_table[i_round]);
        r_key = L_shift(r_key, l_shift_table[i_round]);

        // compression
        string compressed_key = apply_perm(pc2, l_key + r_key, 48);
        compressed_keys[i_round] = compressed_key;
    }

    return compressed_keys;
}

// keys will be same for each block
/*
    @params -> data_block_initial(64 bits of data in binary string form), list
of keys
    @return -> encrypted data_bts
    @Description -> The 64 bit data block passes through 16 rounds of DES and
encrypted block is returned
*/
string encrypt_block(const string &data_block_initial, vector<string>
compressed_keys)
{
    // bitset<64> data_bts(data);
    // apply initial perm on data
```

```
    string data_block = apply_perm(initial_perm, data_block_initial, 64);

    // divide data into two halves : l_half, r_half
    string l_data = data_block.substr(0, 32);
    string r_data = data_block.substr(32, 32);

    for (int i_round = 0; i_round < 16; i_round++)
    {
        string compressed_key = compressed_keys[i_round];

        string r_data_copy = r_data;

        // data operations start
        string r_data_exp = apply_perm(exp_perm, r_data, 48);

        r_data_exp = (bitset<48>(r_data_exp) ^
bitset<48>(compressed_key)).to_string();

        string r_data_s_box = apply_s_box_perm(r_data_exp);

        string r_data_p_box = apply_perm(p_box_perm, r_data_s_box, 32);

        r_data_p_box = (bitset<32>(r_data_p_box) ^
bitset<32>(l_data)).to_string();

        if (i_round != 15)
        {
            r_data = r_data_p_box;

            l_data = r_data_copy;
        }
        else
        {
            l_data = r_data_p_box;
        }
        cout << "Round " << (i_round + 1) << " " << bin_to_hex(l_data) << " "
<< bin_to_hex(r_data) << " " << bin_to_hex(compressed_key) << endl;
    }

    // apply final permute
    string cipher_block = apply_perm(final_perm, (l_data + r_data), 64);

    return bin_to_hex(cipher_block);
}

/*
    @params -> data(input string), key_bin(key in binary string form),
mode(mode of operation (encrypt/decrypt))
    @return -> ecrypted data
    @Description -> takes the input string, encrypts it block by block and
returns the cipher text
*/
string encrypt(string data, const string &key_bin, int mode = MOD_ENCRYPT)
{
    vector<string> compressed_keys = prepare_compressed_keys(key_bin);

    if (mode == MOD_DECRYPT)
    {
        reverse(compressed_keys.begin(), compressed_keys.end());
    }

    string data_bin = hex_to_bin(data);
```

```
    int size_block = 16;

    int n_blocks = (data.size() / size_block);

    string cipher_text = "";

    for (int i_block = 0; i_block < n_blocks; i_block++)
    {
        int starting_idx = i_block * size_block;

        string cur_block = data_bin.substr(starting_idx, size_block * 4);

        string res = encrypt_block(cur_block, compressed_keys);

        cipher_text += res;
    }

    return cipher_text;
}

/*
    @params      : data(input cipher text), key_bin(key in binary string format)
    @return      : original plain text
    @Description: decrypts the original cipher text block by block
*/
string decrypt(const string &data, const string &key_bin)
{
    return encrypt(data, key_bin, MOD_DECRYPT);
}
```

DES Driver function:

```
void DES_Algo(const string &data)
{
    cout <<
"........................................................................" <<
endl;
    cout << "<<<<<<<<<<<<<<<<<<<<   TRIPLE DES ALGORITHM  
>>>>>>>>>>>>>>>>>>>>>>>>>>" << endl;

    // string data = "123456ABCD132536";
    string key1 = "AABB09182736CCDD";
    string key1_bin = hex_to_bin(key1);

    string cipher_text = encrypt(data, key1_bin);

    cout << "Cipher text (DES Algorithm) : " << cipher_text << endl;

    string original_text = decrypt(cipher_text, key1_bin);

    cout << "Original text (DES Algorithm) : " << original_text << endl;
    cout <<
"........................................................................" <<
endl;
}
```

TRIPLE DES driver function:

```
/*
TRIPLE DES ==> EK3(Dk2(Ek1(plaintext)))
decryption ==> Dk1(Ek2(Dk3(plaintext)))
*/
void Triple_DES_Algo(const string &data)
{

    cout <<
"........................................................................" <<
endl;
    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<    TRIPLE DES ALGORITHM  
>>>>>>>>>>>>>>>>>>>>>>>>>>>" << endl;
    string key1 = "AABB09182736CCDD";
    string key2 = "BABB000907667863";
    string key3 = "737374939AAAAAEE";

    string key1_bin = hex_to_bin(key1);
    string key2_bin = hex_to_bin(key2);
    string key3_bin = hex_to_bin(key3);

    string cipher_text = encrypt(decrypt(encrypt(data, key1_bin), key2_bin),
key3_bin);
    cout << "Cipher text = " << cipher_text << endl;

    string original_text = decrypt(encrypt(decrypt(cipher_text, key3_bin),
key2_bin), key1_bin);

    cout << "Original Text = " << original_text << endl;
    cout <<
"........................................................................" <<
endl;
}
```

Driver code:

```
int main()
{
    // key same

    string data = "123456ABCD132536";

    cout << "Original Data  : " << data << endl;

    DES_Algo(data);

    Triple_DES_Algo(data);
}
```

Output:

DES ALGORITHM:

```
Original Data  : 123456ABCD132536
..........................................................................
<<<<<<<<<<<<<<<<<<<< DES ALGORITHM  >>>>>>>>>>>>>>>>>>>>>>>>>>
Round 1 18CA18AD 5A78E394 194CD072DE8C
Round 2 5A78E394 4A1210F6 4568581ABCCE
Round 3 4A1210F6 B8089591 06EDA4ACF5B5
Round 4 B8089591 236779C2 DA2D032B6EE3
Round 5 236779C2 A15A4B87 69A629FEC913
Round 6 A15A4B87 2E8F9C65 C1948E87475E
Round 7 2E8F9C65 A9FC20A3 708AD2DDB3C0
Round 8 A9FC20A3 308BEE97 34F822F0C66D
Round 9 308BEE97 10AF9D37 84BB4473DCCC
Round 10 10AF9D37 6CA6CB20 02765708B5BF
Round 11 6CA6CB20 FF3C485F 6D5560AF7CA5
Round 12 FF3C485F 22A5963B C2C1E96A4BF3
Round 13 22A5963B 387CCDAA 99C31397C91F
Round 14 387CCDAA BD2DD2AB 251B8BC717D0
Round 15 BD2DD2AB CF26B472 3330C5D9A36D
Round 16 19BA9212 CF26B472 181C5D75C66D
Cipher text (DES Algorithm) : C0B7A8D05F3A829C
Round 1 CF26B472 BD2DD2AB 181C5D75C66D
Round 2 BD2DD2AB 387CCDAA 3330C5D9A36D
Round 3 387CCDAA 22A5963B 251B8BC717D0
Round 4 22A5963B FF3C485F 99C31397C91F
Round 5 FF3C485F 6CA6CB20 C2C1E96A4BF3
Round 6 6CA6CB20 10AF9D37 6D5560AF7CA5
Round 7 10AF9D37 308BEE97 02765708B5BF
Round 8 308BEE97 A9FC20A3 84BB4473DCCC
Round 9 A9FC20A3 2E8F9C65 34F822F0C66D
Round 10 2E8F9C65 A15A4B87 708AD2DDB3C0
Round 11 A15A4B87 236779C2 C1948E87475E
Round 12 236779C2 B8089591 69A629FEC913
Round 13 B8089591 4A1210F6 DA2D032B6EE3
Round 14 4A1210F6 5A78E394 06EDA4ACF5B5
Round 15 5A78E394 18CA18AD 4568581ABCCE
Round 16 14A7D678 18CA18AD 194CD072DE8C
Original text (DES Algorithm) : 123456ABCD132536
..........................................................................
```

Triple DES:

```
Original Data  : 123456ABCD132536

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

<<<<<<<<<<<<<<<<<<<<   TRIPLE DES ALGORITHM  >>>>>>>>>>>>>>>>>>>>>>>>>>

Cipher text = 27D52C41CFC8A78C

Original Text = 123456ABCD132536

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
```

Experiment 5

Aim : To implement the working of RSA algorithm.

Theory:

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

An example of asymmetric cryptography :

1.A client (for example browser) sends its public key to the server and requests for some data.

2.The server encrypts the data using client's public key and sends the encrypted data.

3.Client receives this data and decrypts it.

Generating Public key:

```
1. Select two prime no's. Suppose P = 53 and Q = 59.
2. Now First part of the Public key  : n = P*Q = 3127.
3. We also need a small exponent say e :
4. But e Must be an Integer, not a factor of n
5. 1 < e < Φ(n) [Φ(n) is discussed below],
Let us now consider it to be equal to 3.
    Our Public Key is made of n and e
```

Generating Private key:

```
1. We need to calculate Φ(n) :
      Such that Φ(n) = (P-1)(Q-1)
      so,  Φ(n) = 3016
2. Now calculate Private Key, d :
      d = (k*Φ(n) + 1) / e for some integer k
      For k = 2, value of d is 2011.
```

Code:

```
#include <iostream>
#include <string>
#include <vector>
#include <math.h>
#include <algorithm>
#include <sstream>
#include <unordered_map>
#include <queue>
#include <iomanip>
using namespace std;
#define PUBLIC_KEY 65537
#define BLOCK_SIZE 3
```

```cpp
int64_t plain_text = 6882327898788798 79;

int64_t p = 63743, q = 23833;


vector<int64_t> generate_blocks(int64_t plain_text, int size_block)
{
    vector<int64_t> output;
    int dividend = pow(10, size_block);

    while (plain_text)
    {
        int rem = plain_text % dividend;
        output.push_back(rem);
        plain_text /= dividend;
    }

    reverse(output.begin(), output.end());

    return output;
}

/*
    Computes the value of { (base ^ power) % mod }
    Time complexity = O(log(power))
*/

int64_t calc_mod(int64_t base, int64_t power, int64_t mod)
{
    if (power == 0)
        return (1 % mod);

    if (power == 1)
        return (base % mod);

    int mid = (power / 2);

    int res = calc_mod(base, mid, mod);

    res = ((res % mod) * (res % mod)) % mod;

    if (power & 1)
        res = ((res % mod) * (base % mod)) % mod;

    return (res % mod);
}
/*
Main function for encrypting the plaintext blocks
@params:-> plaintext blocks, prime numbers p, q
@return:-> list of encrypted blocks
@Formula to encrypt a block:->
        encrypted_block = (original_block ^ e) % (p * q)
                        where, e = public key
*/

vector<int64_t> encrypt(vector<int64_t> plaintext_blocks, int64_t p, int64_t q)
{
    int64_t n = (p * q);

    // public key
    int64_t e = PUBLIC_KEY;
    // blocks generate kar leta
    vector<int64_t> output_blocks;
```

```
    for (int64_t block : plaintext_blocks)
    {

        // encrypt block
        int64_t encrypted_block = calc_mod(block, e, n);
        // convert it into a single number
        output_blocks.push_back(encrypted_block);
    }

    return output_blocks;
}
/////////////////////////// CODE FOR DECRYPTION OF
FUNCTION ///////////////////////////////////////
/*
    Extended euclid's algorithm for calculating modulo inverse of a number
*/

/*
    @params:-> integers: A, x, B, y, d
    @return:-> map of equations
    @description:->
        Calculate the given set of euations such that A*x + B*y = 1
        Convert the set of equations in form of a map
        Eg: for equation of the form:
            A*x + B*y = d
        Output:
            {
                d: [A, x, B, y]
            }
*/
unordered_map<int64_t, vector<int64_t>> generate_map(int64_t A, int64_t x,
int64_t B, int64_t y, int64_t d)
{
    unordered_map<int64_t, vector<int64_t>> result;
    while (d > 0)
    {
        result[d] = {A, x, B, y};
        A = B;
        B = d;
        y = (A / B);
        d = (A * x - B * y);
    }
    return result;
}

/*
Main algorithm for computing modulo inverse using Euclid's algorithm
*/
pair<int64_t, int64_t> extended_euclid_algo(int64_t A, int64_t x, int64_t B,
int64_t y, int64_t d)
{
    /*
        Reprentation of equations are stored in the variable "expansion"
        eg: Equation:-> 3 * 7 - 5 * 4 = 1
            Entry in map = { 1 : [3, 7, 5, 4] }
        **Used for implementing backward induction
    */
    unordered_map<int64_t, vector<int64_t>> expansion = generate_map(A, x, B,
y, d);

    /*
        store the values of consonents of the variables in an equation
```

```
        eg: Eqn:-> 3 * x - 4 * y = 7
            Entry in consonent :-> {x : 3}, {y : 4}
    */
    unordered_map<int64_t, int64_t> consonent;

    /* keeps track of the next number to be substituted in the main equation */
    queue<int64_t> q;

    /* push the value 1 to the queue: (as the remainder of the last eqn in
euclid algo == 1) */
    q.push(1);
    consonent[1] = 1;

    while (!q.empty())
    {
        int64_t cur_d = q.front();
        q.pop();

        /* take the value of x, y, A, B from the eqn */
        int64_t first_val = expansion[cur_d][0], first_val_consonent =
expansion[cur_d][1];

        int64_t second_val = expansion[cur_d][2], second_val_consonent =
expansion[cur_d][3];

        /* substitute the values and compute consonants */
        consonent[first_val] += (consonent[cur_d] * first_val_consonent);

        consonent[second_val] -= (consonent[cur_d] * second_val_consonent);

        consonent[cur_d] = 0;

        /* Add the values to the queue */
        if (expansion[second_val].size())
            q.push(second_val);
        if (expansion[first_val].size())
            q.push(first_val);
    }

    return {consonent[A], consonent[B]};
}

/*
    Computes (e ^ -1) % t
*/
int64_t mod_inverse_euclid(int64_t e, int64_t t)
{

    // compute A, x, B, y and d

    int64_t A = t;
    int64_t B = e;
    int64_t x = 1;
    int64_t y = t / e;
    int64_t d = (A * x) - (B * y);

    pair<int64_t, int64_t> result = extended_euclid_algo(A, x, B, y, d);

    int64_t y_res = result.second;

    return (y_res < 0 ? (y_res + t) : y_res);
}
```

```
/*
    Main function for decryption
    @params:->  ciphertext block, prime numbers p and q
    @return:-> original text blocks
    @Formula for decrypting a block:
    original_block = { (cipher_block ^ d) % (p * q) }
                    where, d = (e ^ -1) % (p-1 * q-1)
*/
vector<int64_t> decrypt(vector<int64_t> cipher_blocks, int64_t p, int64_t q)
{
    int64_t n = (p * q);

    int64_t t = (p - 1) * (q - 1);

    int64_t e = PUBLIC_KEY;

    int64_t d = mod_inverse_euclid(e, t);

    d %= t;

    vector<int64_t> original_blocks;
    for (int64_t cipher_block : cipher_blocks)
    {
        /* decrypt each block */
        int64_t original_block = calc_mod(cipher_block, d, n);

        /* add the decrypted block to original blocks array*/
        original_blocks.push_back(original_block);
    }
    return original_blocks;
}

void print_vector(vector<int64_t> v, string message)
{
    cout <<
"------------------------------------------------------------------------------
-------------------------" << endl;
    cout << setw(60) << message << endl;
    cout <<
"------------------------------------------------------------------------------
-------------------------" << endl;

    for (int64_t e : v)
        cout << setw(15) << e << "  ";
    cout << endl;
    cout <<
"------------------------------------------------------------------------------
-------------------------" << endl
         << endl;
}
int main()
{
    cout <<
"------------------------------------------------------------------------------
-------------------------" << endl;
    cout << setw(60) << "Original plaintext" << endl;
    cout <<
"------------------------------------------------------------------------------
-------------------------" << endl;

    cout << setw(60) << plain_text << endl;
    cout <<
"------------------------------------------------------------------------------
```

```
-------------------------" << endl;

    vector<int64_t> plaintext_blocks = generate_blocks(plain_text, BLOCK_SIZE);
    print_vector(plaintext_blocks, "Original plaintext blocks(size = 3)");

    // encrypt in block of 3
    vector<int64_t> cipher_blocks = encrypt(plaintext_blocks, p, q);
    print_vector(cipher_blocks, "Encrypted cipher blocks");

    vector<int64_t> original_blocks = decrypt(cipher_blocks, p, q);
    print_vector(original_blocks, "Decrypted plaintext blocks");
}
```

Output:

```
--------------------------------------------------------------------------------
                              Original plaintext
--------------------------------------------------------------------------------
                              688232789878879879
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
                     Original plaintext blocks(size = 3)
--------------------------------------------------------------------------------
      688        232        789        878        879        879
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
                           Encrypted cipher blocks
--------------------------------------------------------------------------------
  199269064  1314500794  1006826272  763160809  1014766002  1014766002
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
                         Decrypted plaintext blocks
--------------------------------------------------------------------------------
      688        232        789        878        879        879
--------------------------------------------------------------------------------
```

Experiment 6

Aim : To implement the working of Deffie-Hellman key exchange algorithm.

Theory:

The Diffie-Hellman algorithm is being used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

•For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables one prime P and G (a primitive root of P) and two private values a and b.

•P and G are both publicly available numbers. Users (say Alice and Bob) pick private values a and b and they generate a key and exchange it publicly, the opposite person received the key and from that generates a secret key after which they have the same secret key to encrypt.

Code:

Bob Machine Code:

```
#include <iostream>
#include <vector>

#include <math.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <sys/types.h>

using namespace std;

/* prime numbers residing in public space */
#define n 997
#define g 7

const int PORT = 5400;

/*
    @params -> base, power, modulo
    @return -> (base ^ power) % modulo
*/
uint64_t pow_mod(uint64_t base, uint64_t power, uint64_t mod)
{
    if (power == 0)
        return 1;

    if (power == 1)
        return base;

    int mid = (power / 2);
```

```
    int res = pow_mod(base, mid, mod);

    res = (res * res) % mod;

    if (power & 1)
        res = (res * base) % mod;

    return (res);
}

/*
    @params -> client_socket
    @return -> void
    @description -> different users share their intermediate secret with each
other(X and Y in our case)
*/
void exchange_keys(int client_socket)
{
    uint64_t x = rand() % 1000;

    uint64_t X = pow_mod(g, x, n);

    send(client_socket, &X, sizeof(X), 0);

    uint64_t Y = -1;

    recv(client_socket, &Y, sizeof(Y), 0);

    // NICE
    uint64_t secret = pow_mod(Y, X, n);

    cout << "The secret is : " << secret << endl;
}

/*
    @params -> client_socket
    @return -> void
    @description -> serves the requests of the clients
*/
void get_served(int client_socket)
{
    char paul_message[2048];

    recv(client_socket, &paul_message, sizeof(paul_message), 0);

    cout << "Paul sent a message : " << string(paul_message) << endl;

    char bob_message[2048] = "Hello Paul";

    send(client_socket, bob_message, sizeof(bob_message), 0);

    exchange_keys(client_socket);
}

int main()
{

    int client_socket = socket(AF_INET, SOCK_STREAM, 0);

    sockaddr_in server_address{};
    server_address.sin_port = htons(PORT);
    server_address.sin_family = AF_INET;
```

```
    server_address.sin_addr.s_addr = INADDR_ANY;

    // connect
    if (connect(client_socket, (sockaddr *)&server_address,
sizeof(server_address)) == -1)
    {
        cerr << "Couldn't connect to server" << endl;
        return -1;
    }

    cout << "Connected to the server" << endl;

    get_served(client_socket);

    // send requests to bob
}
```

Paul Machine Code:

```
#include <iostream>
#include <vector>
#include <string.h>

#include <math.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <unistd.h>

using namespace std;

#define n 997
#define g 7

const int PORT = 5400;

/*
    @params -> base, power, modulo
    @return -> (base ^ power) % modulo
*/
uint64_t pow_mod(uint64_t base, uint64_t power, uint64_t mod)
{
    if (power == 0)
        return 1;

    if (power == 1)
        return base;

    int mid = (power / 2);

    int res = pow_mod(base, mid, mod);

    res = (res * res) % mod;

    if (power & 1)
        res = (res * base) % mod;

    return (res);
```

```
}

/*
    @params -> client_socket
    @return -> void
    @description -> different users share their intermediate secret with each
other(X and Y in our case)
*/
void exchange_keys(int client_socket)
{
    uint64_t y = rand() % 1000;

    uint64_t Y = pow_mod(g, y, n);

    uint64_t X = -1;

    recv(client_socket, &X, sizeof(X), 0);
    // send Y
    send(client_socket, &Y, sizeof(Y), 0);

    uint64_t secret = pow_mod(X, Y, n);

    cout << "The secret is : " << secret << endl;
}

/*
    @params -> client_socket
    @return -> void
    @description -> serves the requests of the clients
*/
void serve_client(int client_socket)
{
    char server_response[] = "Hello bob";

    send(client_socket, server_response, sizeof(server_response), 0);

    char bob_message[2048];

    recv(client_socket, &bob_message, sizeof(bob_message), 0);

    cout << "Bob sent a message : " << string(bob_message) << endl;

    exchange_keys(client_socket);
}

/*
    Driver code
*/
int main()
{

    // initialize a communication
    int server_socket = socket(AF_INET, SOCK_STREAM, 0);

    // struct
    sockaddr_in server_address{};
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(PORT);
    server_address.sin_addr.s_addr = INADDR_ANY;

    // bind
    bind(server_socket, (sockaddr *)&server_address, sizeof(server_address));
```

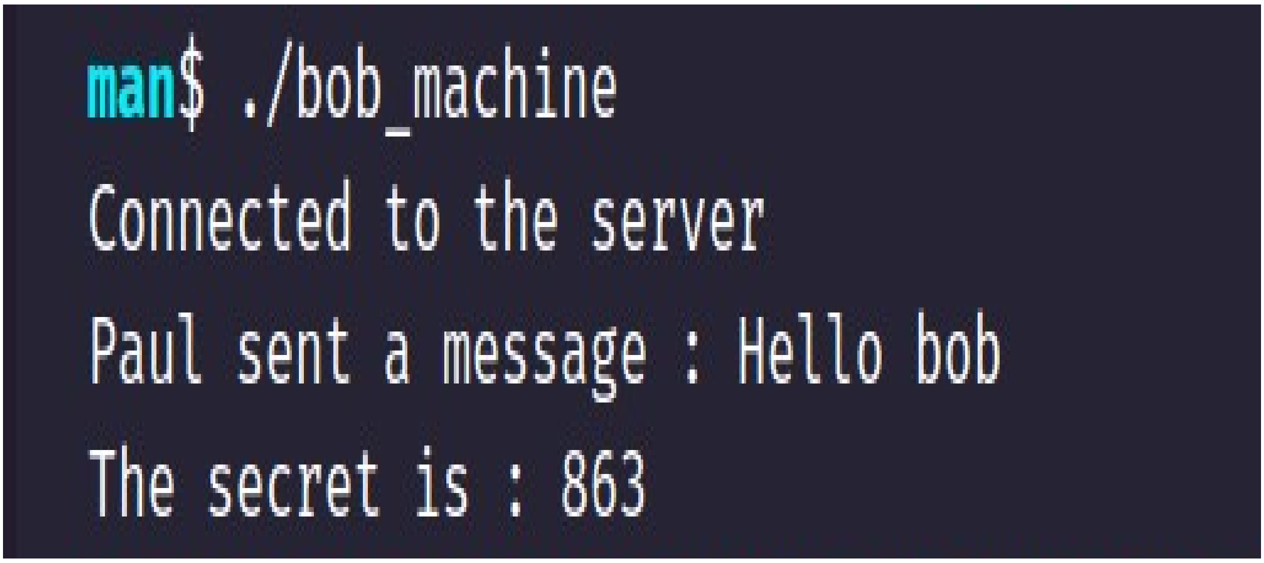
```
    // accept
    listen(server_socket, 5);

    printf("Listening at port %d\n", PORT);

    while (true)
    {
        int client_socket = accept(server_socket, nullptr, nullptr);
        serve_client(client_socket);
        close(client_socket);
    }
}
```

Output:

Bob Machine output:



Paul machine output:

```
man$ ./paul_machine
Listening at port 5400
Bob sent a message : Hello Paul
The secret is : 863
```