MINI PROJECT: Implementation of DES

Aim:-To implement DES Algorithm with GUI and File Handling in Python

Lab Objective:-

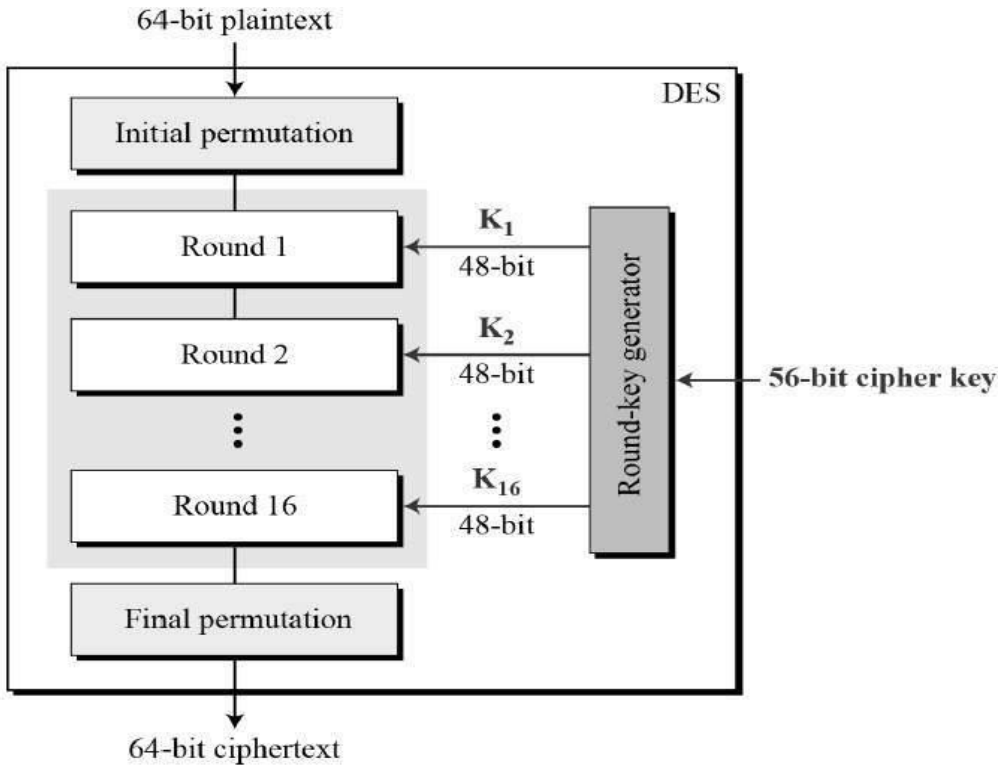
1. To be able to apply the knowledge of symmetric cryptography to implement simple ciphers
2. To analyze and evaluate performance of modern algorithms
3.

Course Objective:-

1. Explore the working principles and utilities of various cryptographic algorithms including secret key cryptography, hashes and message digest and public key algorithm.

Theory:- INTRODUCTION

The Data Encryption Standard (DES) is a symmetric-key block cipher published by the National Institute of Standards and Technology (NIST).



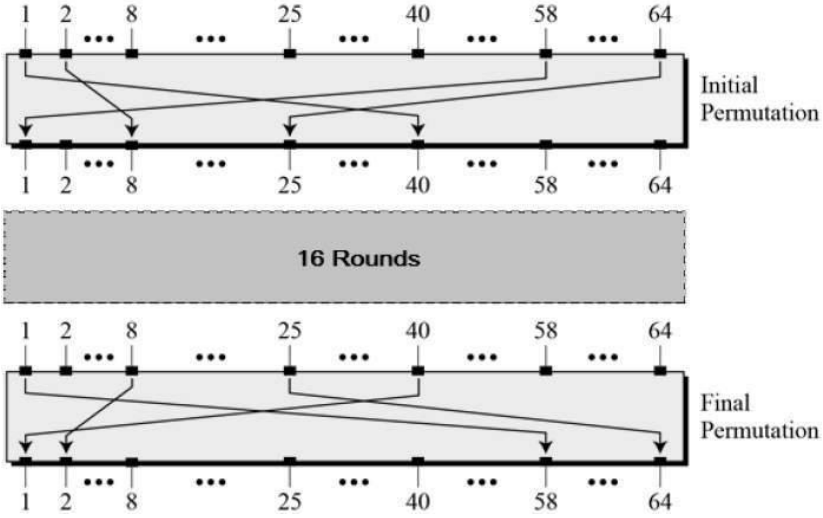
DES is an implementation of a Feistel Cipher. It uses 16 round Feistel structure. The block size is 64-bit. Though, key length is 64-bit, DES has an effective key length of 56 bits, since 8 of the 64 bits of the key are not used by the encryption algorithm (function as check bits only). General Structure of DES is depicted in the following illustration –

Since DES is based on the Feistel Cipher, all that is required to specify DES is −

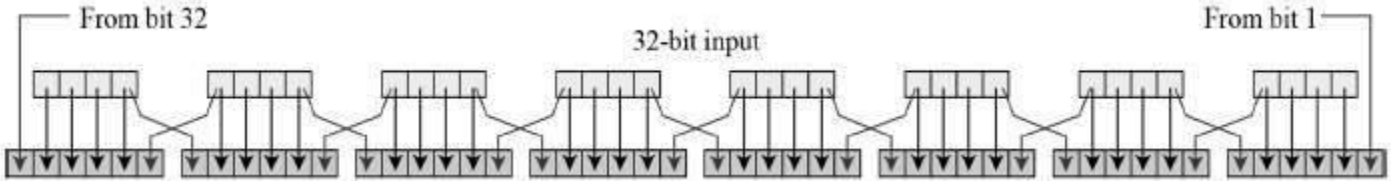
- Round function
- Key schedule
- Any additional processing − Initial and final permutation

Initial and Final Permutation

The initial and final permutations are straight Permutation boxes (P-boxes) that are inverses of each other. They have no cryptography significance in DES. The initial and final permutations are shown as follows –



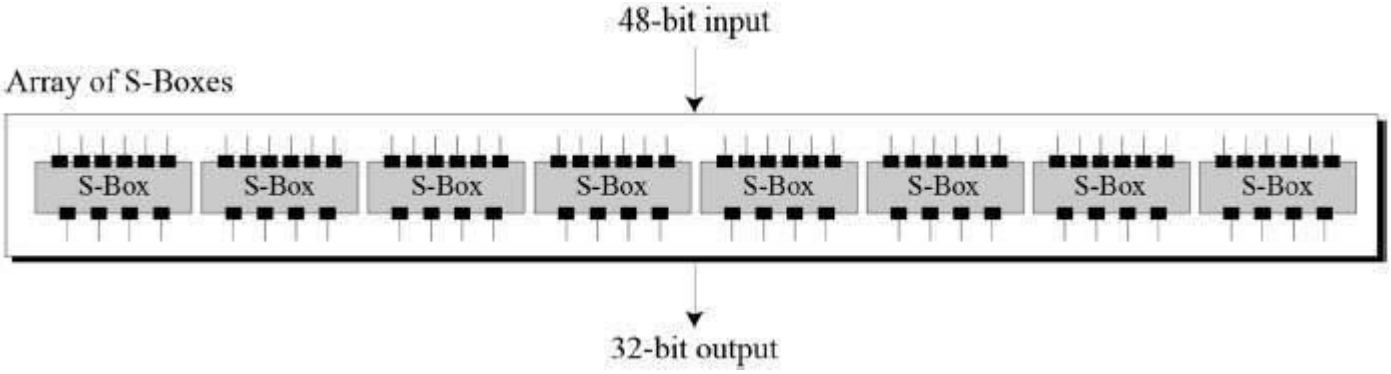
- **Expansion Permutation Box** − Since right input is 32-bit and round key is a 48-bit, we first need to expand right input to 48 bits. Permutation logic is graphically depicted in the following illustration −



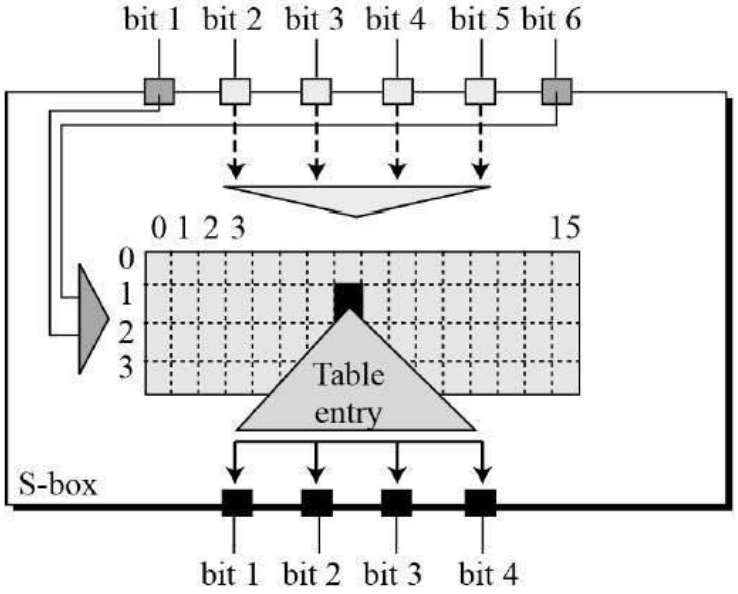
- The graphically depicted permutation logic is generally described as table in DES specification illustrated as shown −

32	01	02	03	04	05
04	05	06	07	08	09
08	09	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	31	31	32	01

- **XOR (Whitener).** − After the expansion permutation, DES does XOR operation on the expanded right section and the round key. The round key is used only in this operation.
- **Substitution Boxes.** − The S-boxes carry out the real mixing (confusion). DES uses 8 S-boxes, each with a 6-bit input and a 4-bit output. Refer the following illustration −



- The S-box rule is illustrated below −



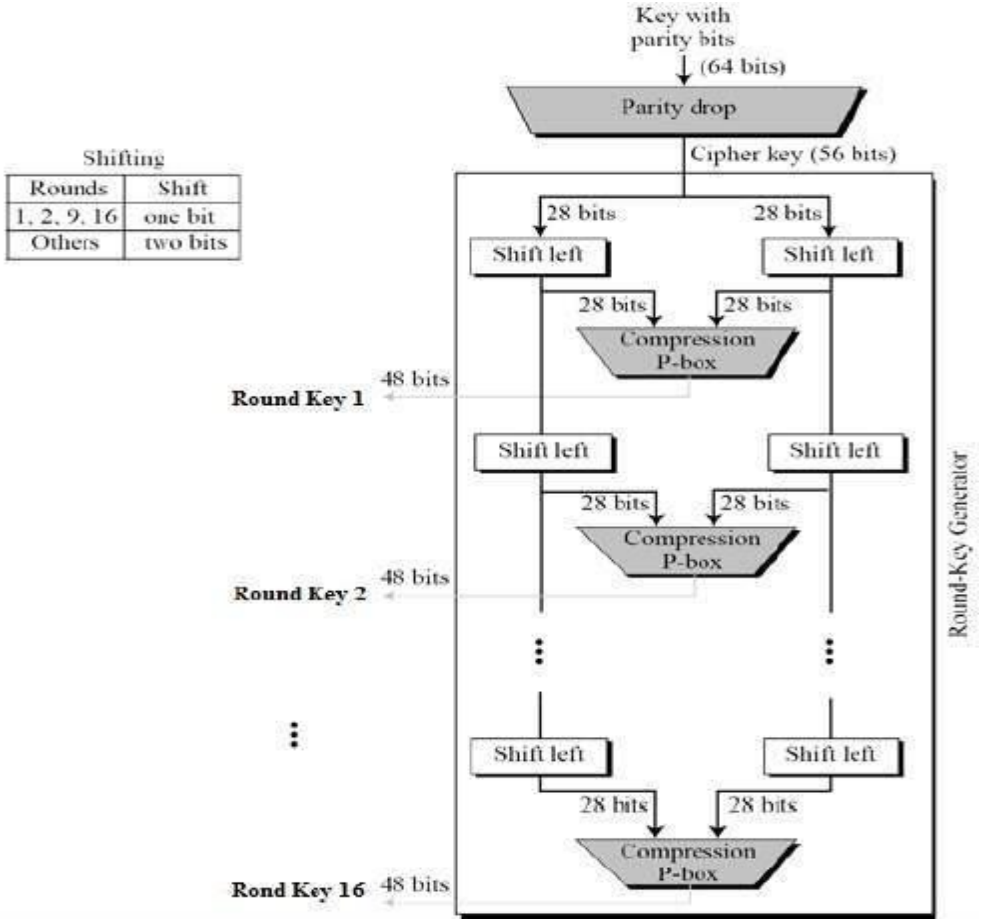
- There are a total of eight S-box tables. The output of all eight s-boxes is then combined in to 32 bit section.

- **Straight Permutation** − The 32 bit output of S-boxes is then subjected to the straight permutation with rule shown in the following illustration:

16	07	20	21	29	12	28	17
01	15	23	26	05	18	31	10
02	08	24	14	32	27	03	09
19	13	30	06	22	11	04	25

Key Generation

The round-key generator creates sixteen 48-bit keys out of a 56-bit cipher key. The process of key generation is depicted in the following illustration −



DES Analysis

The DES satisfies both the desired properties of block cipher. These two properties make cipher very strong.

- **Avalanche effect** − A small change in plaintext results in the very great change in the ciphertext.
- **Completeness** − Each bit of ciphertext depends on many bits of plaintext.

During the last few years, cryptanalysis have found some weaknesses in DES when key selected are weak keys. These keys shall be avoided.

DES has proved to be a very well designed block cipher. There have been no significant cryptanalytic attacks on DES other than exhaustive key search.

Example:

Let **M** be the plain text message **M** = 0123456789ABCDEF, where **M** is in hexadecimal (base 16) format. Rewriting **M** in binary format, we get the 64-bit block of text:

M = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
L = 0000 0001 0010 0011 0100 0101 0110 0111
R = 1000 1001 1010 1011 1100 1101 1110 1111

Let **K** be the hexadecimal key **K** = 133457799BBCDFF1.

K = 00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001

Step 1: Create 16 subkeys, each of which is 48-bits long.

The 64-bit key is permuted according to the following table, PC-1.

we get the 56-bit permutation

K+ = 1111000 0110011 0010101 0101111 0101010 1011001 1001111 0001111

Next, split this key into left and right halves, C_0 and D_0, where each half has 28 bits.

C_0 = 1111000 0110011 0010101 0101111
D_0 = 0101010 1011001 1001111 0001111

We now create sixteen blocks C_n and D_n, $1<=n<=16$. Each pair of blocks C_n and D_n is formed from the previous pair C_{n-1} and D_{n-1}, respectively, for n = 1, 2, ..., 16, using "left shifts" of the previous block. To do a left shift, move each bit one place to the left, except for the first bit, which is cycled to the end of the block.

C_1 = 11100001100110010101011111
D_1 = 10101010110011001111100011110

Create sixteen blocks C_n and D_n,……till C_{16} D_{16}

We now form the keys K_n, for 1<=n<=16, by applying the following permutation table to each of the concatenated pairs C_nD_n. Each pair has 56 bits, but **PC-2** only uses 48 of these.

Example: For the first key we have C_1D_1 = 1110000 1100110 0101010 1011111 1010101 0110011 0011110 0011110

which, after we apply the permutation **PC-2**, becomes

K_1 = 000110 110000 001011 101111 111111 000111 000001 110010

Create sixteen Keys till K_{16}

Step 2: Encode each 64-bit block of data.

1. Applying the initial permutation **IP** to the block of text **M**, given previously, we get

M = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
IP = 1100 1100 0000 0000 1100 1100 1111 1111 1111 0000 1010 1010 1111 0000 1010 1010

Next divide the permuted block **IP** into a left half L_0 of 32 bits, and a right half R_0 of 32 bits.

From **IP**, we get L_0 and R_0

L_0 = 1100 1100 0000 0000 1100 1100 1111 1111
R_0 = 1111 0000 1010 1010 1111 0000 1010 1010

2. We now proceed through 16 iterations, for 1<=n<=16, using a function f which operates on two blocks--a data block of 32 bits and a key K_n of 48 bits--to produce a block of 32 bits. **Let + denote XOR addition, (bit-by-bit addition modulo 2)**. Then for **n** going from 1 to 16 we calculate

$L_n = R_{n-1}$
$R_n = L_{n-1} + f(R_{n-1},K_n)$

For n = 1, we have

K_1 = 000110 110000 001011 101111 111111 000111 000001 110010
$L_1 = R_0$ = 1111 0000 1010 1010 1111 0000 1010 1010
$R_1 = L_0 + f(R_0,K_1)$

3. We calculate $\mathbf{E}(R_0)$ from R_0 as follows:

R_0 = 1111 0000 1010 1010 1111 0000 1010 1010
$\mathbf{E}(R_0)$ = 011110 100001 010101 010101 011110 100001 010101 010101

Next in the ***f*** calculation, we XOR the output $\mathbf{E}(R_{n-1})$ with the key K_n:

$$K_n + \mathbf{E}(R_{n-1}).$$

For K_1 , $\mathbf{E}(R_0)$, we have

K_1 = 000110 110000 001011 101111 111111 000111 000001 110010
$\mathbf{E}(R_0)$ = 011110 100001 010101 010101 011110 100001 010101 010101
$K_1+\mathbf{E}(R_0)$ = 011000 010001 011110 111010 100001 100110 010100 100111.

4. For the first round, we obtain as the output of the eight **S** boxes:

$K_1 + \mathbf{E}(R_0)$ = 011000 010001 011110 111010 100001 100110 010100 100111.

$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$ = 0101 1100 1000 0010 1011 0101 1001 0111

From the output of the eight **S** boxes:

$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$ = 0101 1100 1000 0010 1011 0101 1001 0111

we get

$$f = 0010\ 0011\ 0100\ 1010\ 1010\ 1001\ 1011\ 1011$$

5. $R_1 = L_0 + f(R_0 , K_1)$

= 1100 1100 0000 0000 1100 1100 1111 1111
+ 0010 0011 0100 1010 1010 1001 1011 1011
= 1110 1111 0100 1010 0110 0101 0100 0100

6. In the next round, we will have $L_2 = R_1$, which is the block we just calculated, and then we must calculate $R_2 = L_1 + f(R_1, K_2)$, and so on for 16 rounds. At the end of the sixteenth round we have the blocks L_{16} and R_{16}. We then ***reverse*** the order of the two blocks into the 64-bit block

   $$R_{16}L_{16}$$

   and apply a final permutation $\mathbf{IP}^{-1}$ as defined

7. If we process all 16 blocks using the method defined previously, we get, on the 16th round,

L_{16} = 0100 0011 0100 0010 0011 0010 0011 0100
R_{16} = 0000 1010 0100 1100 1101 1001 1001 0101

We reverse the order of these two blocks and apply the final permutation to

$R_{16}L_{16}$ = 00001010 01001100 11011001 10010101 01000011 01000010 00110010 00110100

IP^{-1} = 10000101 11101000 00010011 01010100 00001111 00001010 10110100 00000101

which in hexadecimal format is

85E813540F0AB405.

This is the encrypted form of **M** = 0123456789ABCDEF: namely, **C** = 85E813540F0AB405.

Decryption

Decryption is simply the inverse of encryption, follwing the same steps as above, but reversing the order in which the subkeys are applied.

Source Code(Python):

```
import subprocess as sp
from tkinter import *

#Initial permut matrix for the datas
IP = [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]

#Initial permut made on the key
CP_1 = [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]
```

```
#Permut applied on shifted key to get Ki+1
CP_2 = [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]

#Expand matrix to get a 48bits matrix of datas to apply the xor with Ki
E = [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]

#SBOX
S_BOX = [

[[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],
]
]

#Permut made after each SBox substitution for each round
P = [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]

#Final permut for datas after the 16 rounds
IP_1 = [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]

#Matrix that determine the shift for each round of keys
SHIFT = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]


def bin2num(n):         #Conversion Binary 2 Number
    return int(n,2)

def num2bin(n):         #Conversion Number 2 Binary
    app=''
    x=bin(n)
    x=x[2:]
    for i in range(4-len(x)):
        app=app+'0'
    x=app+x
    return x
```

```
def permutation(List,string):    #Permutation against Pre-Defined Matrix
    x=''
    for i in List:
        x=x+string[i-1]
        #print(i,string[i-1])
    return x

def kxor(i,j,length):              #String XOR
    app=''
    x=bin(int(i,2)^int(j,2))
    x=x[2:]
    for a in range(length-len(x)):
        app=app+'0'
    x=app+x
    return x

def toBin(i):            #Binary Conversion
    app=''
    #print(ord(i))
    x=bin(ord(i))
    x=x[2:]
    for a in range(8-len(x)):
        app=app+'0'
    x=app+x
    return x

def keyGen(key):      #Key Generator
    bitkey=''
    keys=[]
    finalKeys=[]
    for i in key:
        bitkey=bitkey+toBin(i)
    bitkey=permutation(CP_1,bitkey)
    l,r=bitkey[:28],bitkey[28:]        #C0 D0
    for i in SHIFT:                #C1-16  D1-16
        l=l[i:]+l[:i]
        r=r[i:]+r[:i]
        bitkey=l+r
        keys.append(bitkey)
    for i in keys:
      finalKeys.append(permutation(CP_2,i))     #CP_2 Permutation Keys
    return finalKeys                          #List of Final 16 Keys

def sbox(FRPT):               #SBOX values Generator
    SVAL=''
    for p in range(8):       #p==SBox[p]
        x=FRPT[p*6]+FRPT[(p+1)*6-1]    #x0-3
        y=FRPT[p*6+1:(p+1)*6-1]        #y0-15
        x,y=bin2num(x),bin2num(y)
        sval=S_BOX[p][x][y]
        fsval=num2bin(sval)
```

```
        SVAL=SVAL+fsval
    return SVAL

def toStr(CT):          #Final String Conversion
    TS=''
    for i in range(0,int(len(CT)/8)):
            #print(int(CT[i*8:(i+1)*8-1],2))
            TS=TS+chr(int(CT[i*8:(i+1)*8],2))
    return TS

def run(msg,key,typ):
    app=''
    bitmsg=''
    CT=''
    finalKeys=[]
    if len(key)>8:
        return "Enter 8 Character Key"
    elif len(key)<=8:
        for a in range(8-len(key)):
            app=app+'0'
        key=app+key
        finalKeys=keyGen(key)      #KeyGeneration
        if typ==2:
            finalKeys=finalKeys[::-1]
        #print(finalKeys)
        app=''
        if not(len(msg)%8==0) or len(msg)==0:
            for a in range(8-(len(msg)%8)):
                app=app+'0'
        msg=app+msg
        for i in range(int(len(msg)/8)):  #Blocks of 8 Characters
            for j in range(8):   #8 Characters
                bitmsg=bitmsg+toBin(msg[(i*8)+j])
            #print(bitmsg,len(bitmsg))
            bitmsg=permutation(IP,bitmsg)   #Initial Permutation
            #print("AFTer IP",bitmsg,len(bitmsg))
            LPT=bitmsg[:32]
            RPT=bitmsg[32:]
            #print(LPT,len(LPT),RPT,len(RPT))
            FRPT=RPT[:]
            for k in range(0,16):
                #print(LPT,RPT)
                FRPT=permutation(E,RPT)
                FRPT=kxor(FRPT,finalKeys[k],48)
                FRPT=sbox(FRPT)
                FRPT=permutation(P,FRPT)   #P Permutation
                FRPT=kxor(FRPT,LPT,32)
                LPT=RPT[:]
                RPT=FRPT[:]
                #print(LPT,len(LPT),RPT,len(RPT))
            bitmsg=RPT+LPT
```

```
            bitmsg=permutation(IP_1,bitmsg)   #Final Permutation
            #print("AFTer IP_1",bitmsg,len(bitmsg))
            CT=CT+bitmsg
            bitmsg="
        #print(CT)
        FCT=toStr(CT)
        return FCT


def encrypt(msg,key):        #Encryption Call
    return run(msg,key,1)

def decrypt(msg,key):        #Decryption Call
    return run(msg,key,2)


def Encry():             #GUI Encryption passing
    t="
    x=e1.get()
    file = open(x, 'r',encoding='utf-8')
    msg=file.read()
    file1=open('output.txt','w',encoding='utf-8')
    key=e2.get()
    if msg=="":
        FCT='Enter Message'
    elif key=="":
        FCT='Enter Key'
    else:
        FCT=encrypt(msg,key)
        file1.write("CT:")
        file1.write(FCT)
        FCT="Cipher Text: "+FCT
    if len(FCT)>44:
        z=0
        for i in range(int(len(FCT)/44)):
            t=t+FCT[i*44:(i+1)*44]
            t=t+'\n'
            z=z+1
        t=t+FCT[z*44:]
    m.config(text=t)
    print(FCT)
    file.close()
    file1.close()

    programName = "notepad.exe"
    fileName = "output.txt"
    sp.Popen([programName, fileName])

def Decry():         #GUI Decryption passing
    t="
    y=e1.get()
```

```
    file = open(y, 'r',encoding='utf-8')
    msg=file.read()
    msg=msg[3:]
    file1=open('DEoutput.txt','w',encoding='utf-8')
    key=e2.get()
    if msg=='':
        DCT='Enter Message'
    elif key=='':
        DCT='Enter Key'
    else:
        DCT=decrypt(msg,key)
        DCT=DCT[8:]
        file1.write(DCT)
        DCT="Plain Text: "+DCT
        if len(DCT)>44:
            z=0
            for i in range(int(len(DCT)/44)):
                t=t+DCT[i*44:(i+1)*44]
                t=t+'-\n'
                z=z+1
            t=t+DCT[z*44:]
    m.config(text=t)
    print(DCT)
    file.close()
    file1.close()

    programName = "notepad.exe"
    fileName = "DEoutput.txt"
    sp.Popen([programName, fileName])

def Exit():         #Exit Function
    master.destroy()

#GUI Code
master = Tk()
master.config(bg='yellow',bd=3)
master.title('DES')
master.minsize(300, 300)
master.maxsize(300, 300)
space=Canvas(master, width=300, height=10,bg='yellow', bd=0, highlightthickness=0, relief='ridge')
space.pack()
h = Canvas(master, width=300, height=50)
h.pack()
l1=Label(h, text='DES',font=("Courier", 32,'underline'),bg='turquoise',relief="solid",bd=2)
l1.pack()
space=Canvas(master, width=300, height=10,bg='yellow', bd=0, highlightthickness=0, relief='ridge')
space.pack()
con = Canvas(master, width=300, height=100,bg='yellow',relief="solid",bd=2)
con.pack()
Label(con, text='File-Name',width=15,bg='orange',bd=1,relief='solid').grid(row=1,padx=5, pady=10)
Label(con, text='Key',width=15,bg='orange',bd=1,relief='solid').grid(row=2,padx=5, pady=10)
```

```
e1 = Entry(con,width=20)
e2 = Entry(con,width=20)
e1.grid(row=1, column=1,padx=10, pady=10)
e2.grid(row=2, column=1,padx=10, pady=10)
space=Canvas(master, width=300, height=10,bg='yellow', bd=0, highlightthickness=0, relief='ridge')
space.pack()
t = Canvas(master, width=300, height=50,relief="solid",bd=2)
t.pack()
m=Label(t, text='Encrypted/Decrypted Text Here!',width=39)
m.grid(row=3,padx=10, pady=10)
space=Canvas(master, width=300, height=10,bg='yellow', bd=0, highlightthickness=0, relief='ridge')
space.pack()
bt = Canvas(master, width=300, height=50,bg='blue',relief="solid",bd=2)
bt.pack()
b1=Button(bt,text='Encrypt',width=10,command=Encry)
b2=Button(bt,text='Decrypt',width=10,command=Decry)
b3=Button(bt,text='Exit',width=10,command=Exit)
b1.grid(row=0,column=0,padx=5, pady=5)
b2.grid(row=0,column=1,padx=5, pady=5)
b3.grid(row=0,column=3,padx=5, pady=5)
footer = Canvas(master, width=300, height=15, bd=2, highlightthickness=0, relief='solid')
footer.pack(side='bottom')
footer.create_text(100,10,fill="darkblue",font="Times 8 italic bold", text="       Copyright: \u00A9 2019
Abhishek Soni",anchor='w')
mainloop()
```

Lab Outcomes:-

1. Apply the knowledge of symmetric cryptography to implement simple ciphers.
2. Analyze and evaluate performance of modern algorithms.

Course Outcome:-

1. Understand compare and apply different encryption and decryption techniques to olve the problem related to confidentiality and authentication.

Result and Discussions:

Successful implementation and deployment of DES Mini-application.

Conclusion:

In this Project we made a DES Mini-application with optimal GUI and handled the user text based on different files also with consideration of their encoding standards and predefined formats.