IS-1 AND XOR Encryption

Code

#include <iostream.h>

//using namespace std;

#include <stdio.h>

#include <conio.h>

#include <string.h>

#include <stdlib.h>

void main()

{

//clrscr();

char str[]="HELLOWORLD";

char str1[11];

char str2[11];

int i,len;

len = strlen(str);

for(i=0;i<len;i++)

{

str1[i]=str[i] & 127;

cout<<str1[i];

}

cout<<"\n";

for(i=0;i<len;i++)

{

str2[i] = str[i] ^ 127;

cout<<str2[i];

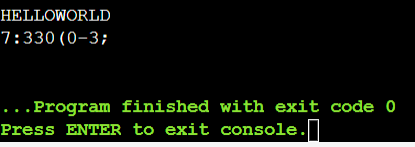
}

cout<<"\n";

getch();

}

Output



IS-2 Transposition Technique- Columnar

Code

import math

key = "HACK"

# Encryption

def encryptMessage(msg):

    cipher = ""

    # track key indices

    k\_indx = 0

    msg\_len = float(len(msg))

    msg\_lst = list(msg)

    key\_lst = sorted(list(key))

    # calculate column of the matrix

    col = len(key)

    # calculate maximum row of the matrix

    row = int(math.ceil(msg\_len / col))

    # add the padding character '\_' in empty

    # the empty cell of the matix

    fill\_null = int((row \* col) - msg\_len)

    msg\_lst.extend('\_' \* fill\_null)

    # create Matrix and insert message and

    # padding characters row-wise

    matrix = [msg\_lst[i: i + col]

              for i in range(0, len(msg\_lst), col)]

    # read matrix column-wise using key

    for \_ in range(col):

        curr\_idx = key.index(key\_lst[k\_indx])

        cipher += ''.join([row[curr\_idx]

                          for row in matrix])

        k\_indx += 1

    return cipher

# Decryption

def decryptMessage(cipher):

    msg = ""

    # track key indices

    k\_indx = 0

    # track msg indices

    msg\_indx = 0

    msg\_len = float(len(cipher))

    msg\_lst = list(cipher)

    # calculate column of the matrix

    col = len(key)

    # calculate maximum row of the matrix

    row = int(math.ceil(msg\_len / col))

    # convert key into list and sort

    # alphabetically so we can access

    # each character by its alphabetical position.

    key\_lst = sorted(list(key))

    # create an empty matrix to

    # store deciphered message

    dec\_cipher = []

    for \_ in range(row):

        dec\_cipher += [[None] \* col]

    # Arrange the matrix column wise according

    # to permutation order by adding into new matrix

    for \_ in range(col):

        curr\_idx = key.index(key\_lst[k\_indx])

        for j in range(row):

            dec\_cipher[j][curr\_idx] = msg\_lst[msg\_indx]

            msg\_indx += 1

        k\_indx += 1

    # convert decrypted msg matrix into a string

    try:

        msg = ''.join(sum(dec\_cipher, []))

    except TypeError:

        raise TypeError("This program cannot",

                        "handle repeating words.")

    null\_count = msg.count('\_')

    if null\_count > 0:

        return msg[: -null\_count]

    return msg

# Driver Code

msg = "WEARETHEBEST"

cipher = encryptMessage(msg)

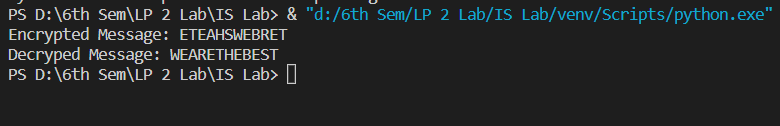
print("Encrypted Message: {}".

               format(cipher))

print("Decryped Message: {}".

       format(decryptMessage(cipher)))

Output



IS-3 DES Algorithm

Code

# Hexadecimal to binary conversion

def hex2bin(s):

    mp = {'0' : "0000",

        '1' : "0001",

        '2' : "0010",

        '3' : "0011",

        '4' : "0100",

        '5' : "0101",

        '6' : "0110",

        '7' : "0111",

        '8' : "1000",

        '9' : "1001",

        'A' : "1010",

        'B' : "1011",

        'C' : "1100",

        'D' : "1101",

        'E' : "1110",

        'F' : "1111" }

    bin = ""

    for i in range(len(s)):

        bin = bin + mp[s[i]]

    return bin

# Binary to hexadecimal conversion

def bin2hex(s):

    mp = {"0000" : '0',

        "0001" : '1',

        "0010" : '2',

        "0011" : '3',

        "0100" : '4',

        "0101" : '5',

        "0110" : '6',

        "0111" : '7',

        "1000" : '8',

        "1001" : '9',

        "1010" : 'A',

        "1011" : 'B',

        "1100" : 'C',

        "1101" : 'D',

        "1110" : 'E',

        "1111" : 'F' }

    hex = ""

    for i in range(0,len(s),4):

        ch = ""

        ch = ch + s[i]

        ch = ch + s[i + 1]

        ch = ch + s[i + 2]

        ch = ch + s[i + 3]

        hex = hex + mp[ch]

    return hex

# Binary to decimal conversion

def bin2dec(binary):

    binary1 = binary

    decimal, i, n = 0, 0, 0

    while(binary != 0):

        dec = binary % 10

        decimal = decimal + dec \* pow(2, i)

        binary = binary//10

        i += 1

    return decimal

# Decimal to binary conversion

def dec2bin(num):

    res = bin(num).replace("0b", "")

    if(len(res)%4 != 0):

        div = len(res) / 4

        div = int(div)

        counter =(4 \* (div + 1)) - len(res)

        for i in range(0, counter):

            res = '0' + res

    return res

# Permute function to rearrange the bits

def permute(k, arr, n):

    permutation = ""

    for i in range(0, n):

        permutation = permutation + k[arr[i] - 1]

    return permutation

# shifting the bits towards left by nth shifts

def shift\_left(k, nth\_shifts):

    s = ""

    for i in range(nth\_shifts):

        for j in range(1,len(k)):

            s = s + k[j]

        s = s + k[0]

        k = s

        s = ""

    return k

# calculating xow of two strings of binary number a and b

def xor(a, b):

    ans = ""

    for i in range(len(a)):

        if a[i] == b[i]:

            ans = ans + "0"

        else:

            ans = ans + "1"

    return ans

# Table of Position of 64 bits at initial level: Initial Permutation Table

initial\_perm = [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]

# Expansion D-box Table

exp\_d = [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 ]

# Straight Permutation Table

per = [ 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 ]

# S-box Table

sbox = [[[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] ] ]

# Final Permutation Table

final\_perm = [ 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 ]

def encrypt(pt, rkb, rk):

    pt = hex2bin(pt)

    # Initial Permutation

    pt = permute(pt, initial\_perm, 64)

    print("After initial permutation", bin2hex(pt))

    # Splitting

    left = pt[0:32]

    right = pt[32:64]

    for i in range(0, 16):

        # Expansion D-box: Expanding the 32 bits data into 48 bits

        right\_expanded = permute(right, exp\_d, 48)

        # XOR RoundKey[i] and right\_expanded

        xor\_x = xor(right\_expanded, rkb[i])

        # S-boxex: substituting the value from s-box table by calculating row and column

        sbox\_str = ""

        for j in range(0, 8):

            row = bin2dec(int(xor\_x[j \* 6] + xor\_x[j \* 6 + 5]))

            col = bin2dec(int(xor\_x[j \* 6 + 1] + xor\_x[j \* 6 + 2] + xor\_x[j \* 6 + 3] + xor\_x[j \* 6 + 4]))

            val = sbox[j][row][col]

            sbox\_str = sbox\_str + dec2bin(val)

        # Straight D-box: After substituting rearranging the bits

        sbox\_str = permute(sbox\_str, per, 32)

        # XOR left and sbox\_str

        result = xor(left, sbox\_str)

        left = result

        # Swapper

        if(i != 15):

            left, right = right, left

        print("Round ", i + 1, " ", bin2hex(left), " ", bin2hex(right), " ", rk[i])

    # Combination

    combine = left + right

    # Final permutation: final rearranging of bits to get cipher text

    cipher\_text = permute(combine, final\_perm, 64)

    return cipher\_text

pt = "123456ABCD132536"

key = "AABB09182736CCDD"

# Key generation

# --hex to binary

key = hex2bin(key)

# --parity bit drop table

keyp = [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 ]

# getting 56 bit key from 64 bit using the parity bits

key = permute(key, keyp, 56)

# Number of bit shifts

shift\_table = [1, 1, 2, 2,

                2, 2, 2, 2,

                1, 2, 2, 2,

                2, 2, 2, 1 ]

# Key- Compression Table : Compression of key from 56 bits to 48 bits

key\_comp = [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 ]

# Splitting

left = key[0:28] # rkb for RoundKeys in binary

right = key[28:56] # rk for RoundKeys in hexadecimal

rkb = []

rk = []

for i in range(0, 16):

    # Shifting the bits by nth shifts by checking from shift table

    left = shift\_left(left, shift\_table[i])

    right = shift\_left(right, shift\_table[i])

    # Combination of left and right string

    combine\_str = left + right

    # Compression of key from 56 to 48 bits

    round\_key = permute(combine\_str, key\_comp, 48)

    rkb.append(round\_key)

    rk.append(bin2hex(round\_key))

print("Encryption")

cipher\_text = bin2hex(encrypt(pt, rkb, rk))

print("Cipher Text : ",cipher\_text)

print("Decryption")

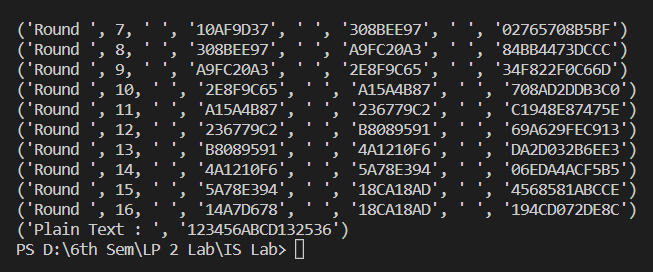
rkb\_rev = rkb[::-1]

rk\_rev = rk[::-1]

text = bin2hex(encrypt(cipher\_text, rkb\_rev, rk\_rev))

print("Plain Text : ",text)

Output



IS- 4 AES Algorithm

Code

import hashlib

from base64 import b64decode, b64encode

from Crypto import Random

from Crypto.Cipher import AES

class AESCipher(object):

    def \_\_init\_\_(self, key):

        self.block\_size = AES.block\_size

        self.key = hashlib.sha256(key.encode()).digest()

    def encrypt(self, plain\_text):

        plain\_text = self.\_\_pad(plain\_text)

        iv = Random.new().read(self.block\_size)

        cipher = AES.new(self.key, AES.MODE\_CBC, iv)

        encrypted\_text = cipher.encrypt(plain\_text.encode())

        return b64encode(iv + encrypted\_text).decode("utf-8")

    def decrypt(self, encrypted\_text):

        encrypted\_text = b64decode(encrypted\_text)

        iv = encrypted\_text[:self.block\_size]

        cipher = AES.new(self.key, AES.MODE\_CBC, iv)

        plain\_text = cipher.decrypt(encrypted\_text[self.block\_size:]).decode("utf-8")

        return self.\_\_unpad(plain\_text)

    def \_\_pad(self, plain\_text):

        number\_of\_bytes\_to\_pad = self.block\_size - len(plain\_text) % self.block\_size

        ascii\_string = chr(number\_of\_bytes\_to\_pad)

        padding\_str = number\_of\_bytes\_to\_pad \* ascii\_string

        padded\_plain\_text = plain\_text + padding\_str

        return padded\_plain\_text

    @staticmethod

    def \_\_unpad(plain\_text):

        last\_character = plain\_text[len(plain\_text) - 1:]

        return plain\_text[:-ord(last\_character)]

key = input("Enter Key: ")

aes = AESCipher(key)

message = input("Enter message to encrypt: ")

encryptedMessage = aes.encrypt(message)

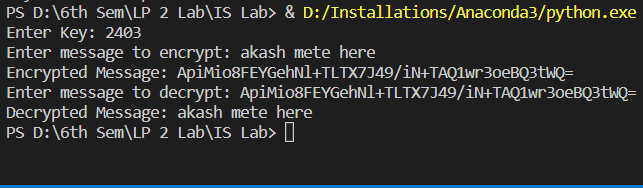
print("Encrypted Message:", encryptedMessage)

message = input("Enter message to decrypt: ")

decryptedMessage = aes.decrypt(message)

print("Decrypted Message:", decryptedMessage)

Output



IS-5 RSA Algorithm

Code

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1\_OAEP

import binascii

msg = (input("Enter message to encrypt and decrypt"))

msg = bytes(msg, 'utf-8')

keyPair = RSA.generate(3072)

pubKey = keyPair.publickey()

print(f"Public key:  (n={hex(pubKey.n)}, e={hex(pubKey.e)})")

pubKeyPEM = pubKey.exportKey()

print(pubKeyPEM.decode('ascii'))

print(f"Private key: (n={hex(pubKey.n)}, d={hex(keyPair.d)})")

privKeyPEM = keyPair.exportKey()

print(privKeyPEM.decode('ascii'))

# msg = input()

encryptor = PKCS1\_OAEP.new(pubKey)

encrypted = encryptor.encrypt(msg)

print("Encrypted:", binascii.hexlify(encrypted))

decryptor = PKCS1\_OAEP.new(keyPair)

decrypted = decryptor.decrypt(encrypted)

print('Decrypted:', decrypted)

Output

