CRYPTOGRAPHY FUNDAMENTALS LAB

EXPERIMENT - 3

- **NAME: - ANSHIL SETH**
- **REG NO. : - 18BCI0173**
- **SLOT: - L15+ L16**
- **FACULTY: - PROF. RAMANI S.**

QUESTION: -

Create text file sample.txt; encrypt the file sample.txt using asymmetric key. The first time when code is run, a folder is created. You will be asked to enter a key. The key use dis "InfoSec". An encrypted file is now created in the same location as the plain text file with the name "sample1.txt"andseesdifferenceinfile.And also perform the Decryption also.(Use AES algorithm)

CODE: - (In PYTHON)

```
import sys

import os

s_box=

 ( 0x63,0x7C,0x77,0x7B,0xF2,0x6B,0x6F,0xC5,0x30,0x01,0x67,0x2B,0xFE, 0xD7, 0xAB, 0x76,
0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72,
0xC0, 0xB7,0xFD, 0x93,0x26,0x36, 0x3F,0xF7,0xCC, 0x34,0xA5, 0xE5,0xF1,0x71, 0xD8, 0x31,
0x15, 0x04, 0xC7,0x23, 0xC3,0x18, 0x96, 0x05,0x9A, 0x07,0x12, 0x80,0xE2, 0xEB, 0x27, 0xB2,
0x75, 0x09,0x83,0x2C,0x1A,0x1B,0x6E,0x5A,0xA0,0x52,0x3B,0xD6, 0xB3,0x29, 0xE3, 0x2F,
0x84, 0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C,
0x58, 0xCF, 0xD0, 0xEF, 0xAA, 0xFB, 0x43,0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F,0x50, 0x3C,
0x9F, 0xA8,0x51,0xA3, 0x40, 0x8F,0x92, 0x9D,0x38,0xF5,0xBC, 0xB6, 0xDA,0x21,0x10, 0xFF,
0xF3, 0xD2, 0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64,
0x5D, 0x19, 0x73, 0x60,0x81,0x4F,0xDC,0x22,0x2A,0x90,0x88,0x46,0xEE,0xB8,0x14,0xDE,
0x5E, 0x0B, 0xDB, 0xE0,0x32,0x3A,0x0A,0x49,0x06,0x24,0x5C, 0xC2,0xD3,0xAC, 0x62,0x91,
0x95, 0xE4, 0x79, 0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA,
0x65, 0x7A, 0xAE, 0x08, 0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74,
0x1F, 0x4B, 0xBD, 0x8B, 0x8A, 0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35,
0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E, 0xE1,0xF8, 0x98, 0x11,0x69, 0xD9,0x8E,0x94,0x9B, 0x1E,
0x87, 0xE9,0xCE, 0x55, 0x28, 0xDF, 0x8C,0xA1, 0x89,0x0D, 0xBF,0xE6, 0x42, 0x68,0x41,0x99,
0x2D,0x0F,0xB0, 0x54, 0xBB, 0x16, )
```

inv_s_box= (0x52, 0x09,0x6A, 0xD5,0x30, 0x36, 0xA5,0x38, 0xBF,0x40, 0xA3,0x9E, 0x81, 0xF3, 0xD7, 0xFB, 0x7C,0xE3, 0x39,0x82,0x9B,0x2F,0xFF,0x87, 0x34,0x8E,0x43,0x44,0xC4, 0xDE, 0xE9, 0xCB, 0x54,0x7B, 0x94, 0x32,0xA6, 0xC2,0x23,0x3D, 0xEE,0x4C, 0x95,0x0B,0x42,0xFA,0xC3,0x4E,0x08,0x2E,0xA1,0x66,0x28,0xD9,0x24,0xB2,0x76,0x5B,0xA2,0x49,0x6D, 0x8B, 0xD1, 0x25, 0x72,0xF8, 0xF6, 0x64,0x86, 0x68, 0x98,0x16,0xD4, 0xA4,0x5C,0xCC, 0x5D, 0x65, 0xB6, 0x92, 0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84, 0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06, 0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B, 0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE,0xF0,0xB4,0xE6,0x73,0x96,0xAC,0x74,0x22,0xE7,0xAD,0x35,0x85,0xE2,0xF9,0x37,0xE8,0x1C, 0x75, 0xDF, 0x6E, 0x47, 0xF1,0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62,0x0E, 0xAA, 0x18, 0xBE, 0x1B, 0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4, 0x1F, 0xDD,0xA8, 0x33, 0x88,0x07, 0xC7,0x31,0xB1, 0x12,0x10, 0x59,0x27, 0x80, 0xEC, 0x5F, 0x60, 0x51,0x7F, 0xA9, 0x19, 0xB5, 0x4A,0x0D, 0x2D, 0xE5, 0x7A, 0x9F,0x93, 0xC9, 0x9C, 0xEF, 0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61, 0x17, 0x2B, 0x04, 0x7E, 0xBA,0x77, 0xD6, 0x26,0xE1, 0x69,0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D,)

def sub_bytes(s): fori in range(4): forj in range(4): s[i][j] =s_box[s[i][j]]

definv_sub_bytes(s): fori in range(4): forj in range(4): s[i][j] =inv_s_box[s[i][j]]

def shift_rows(s): s[0][1], s[1][1],s[2][1], s[3][1]= s[1][1], s[2][1],s[3][1], s[0][1] s[0][2], s[1][2],s[2][2], s[3][2]= s[2][2], s[3][2],s[0][2], s[1][2] s[0][3], s[1][3],s[2][3], s[3][3]= s[3][3], s[0][3],s[1][3], s[2][3]

def inv_shift_rows(s): s[0][1], s[1][1],s[2][1], s[3][1]= s[3][1], s[0][1],s[1][1], s[2][1] s[0][2], s[1][2],s[2][2], s[3][2]= s[2][2], s[3][2],s[0][2], s[1][2] s[0][3], s[1][3],s[2][3], s[3][3]= s[1][3], s[2][3],s[3][3], s[0][3]

def add_round_key(s, k): fori in range(4): forj in range(4): s[i][j] ^= k[i][j]

x time= lambda a:(((a<< 1) ^0x1B) & 0xFF) if (a&0x80) else (a<<1)

def mix_single_column(a): t = a[0]^ a[1]^ a[2]^ a[3] u = a[0] a[0] ^= t ^xtime(a[0] ^a[1]) a[1] ^= t ^xtime(a[1] ^a[2]) a[2] ^= t ^xtime(a[2] ^a[3]) a[3] ^= t ^xtime(a[3] ^u)

def mix_columns(s): fori in range(4): mix_single_column(s[i])

def inv_mix_columns(s): fori in range(4): u= x time(xtime(s[i][0]^ s[i][2])) v= x time(xtime(s[i][1]^ s[i][3])) s[i][0]^= u s[i][1]^= v s[i][2]^= u

s[i][3]^= v mix_columns(s)

r_con= (0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36, 0x6C, 0xD8, 0xAB, 0x4D, 0x9A, 0x2F, 0x5E, 0xBC, 0x63, 0xC6, 0x97, 0x35, 0x6A, 0xD4, 0xB3, 0x7D, 0xFA, 0xEF, 0xC5, 0x91, 0x39,)

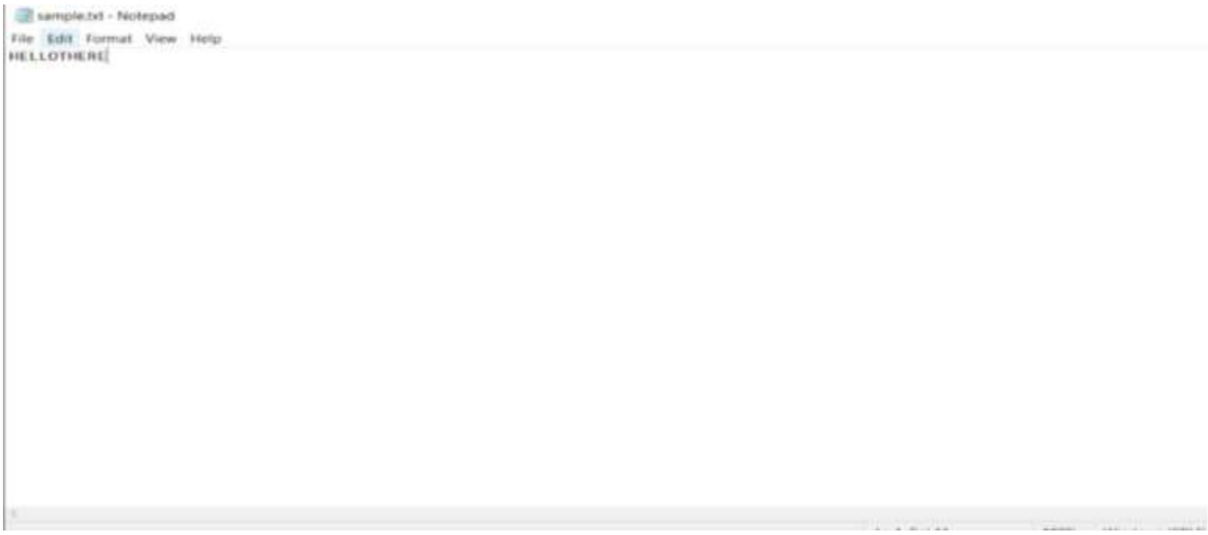
def bytes2matrix(text): return[list(text[i:i + 4])

```
for i inrange(0, len(text),4)]

def matrix2bytes(matrix):

return bytes(sum(matrix, []))

def x or_bytes(a, b):

returnbytes(i^ j fori, jin zip(a, b))

definc_bytes(a): out = list(a) fori in reversed(range(len(out))): if out[i] == 0xFF: out[i] = 0 else:
out[i] += 1 break returnbytes(out)

defpad(plaintext): padding_len= 16 -(len(plaintext) % 16) padding = bytes([padding_len]
*padding_len) returnplaintext + padding

defunpad(plaintext): padding_len= plaintext[-1] assertpadding_len> 0 message, padding =
plaintext[:-padding_len],plaintext[-padding_len:] assertall(p == padding_len forpin padding)
returnmessage

defsplit_blocks(message, block_size=16): assertlen(message) % block_size ==0
return[message[i:i+ 16] fori in range(0, len(message), block_size)]

classAES: rounds_by_key_size = {16: 10, 24:12, 32: 14} def __init__(self,master_key): assert
len(master_key) in AES.rounds_by_key_size self.n_rounds=
AES.rounds_by_key_size[len(master_key)] self._key_matrices= self._expand_key(master_key)
def _expand_key(self,master_key): key_columns= bytes2matrix(master_key) iteration_size =
len(master_key) //4 columns_per_iteration= len(key_columns) i= 1 while len(key_columns) <
(self.n_rounds+ 1) * 4: word= list(key_columns[-1])

if len(key_columns) % iteration_size == 0: word.append(word.pop(0)) word= [s_box[b] forbin
word] word[0]^= r_con[i] i += 1 elif len(master_key) ==32 andlen(key_columns) %
iteration_size ==4: word= [s_box[b] forbin word]

word= xor_bytes(word,key_columns[-iteration_size]) key_columns.append(word)

return[key_columns[4 * i: 4* (i+ 1)] fori in range(len(key_columns)// 4)] def encrypt_block(self,
plaintext): assert len(plaintext) ==16 plain_state= bytes2matrix(plaintext)
add_round_key(plain_state,self._key_matrices[0])

fori in range(1,self.n_rounds): sub_bytes(plain_state) shift_rows(plain_state)
mix_columns(plain_state) add_round_key(plain_state, self._key_matrices[i])
sub_bytes(plain_state) shift_rows(plain_state) add_round_key(plain_state,self._key_matrices[-
1]) returnmatrix2bytes(plain_state) def decrypt_block(self, ciphertext): assert len(ciphertext)
==16 cipher_state= bytes2matrix(ciphertext) add_round_key(cipher_state,self._key_matrices[-
1]) inv_shift_rows(cipher_state) inv_sub_bytes(cipher_state) fori in range(self.n_rounds- 1, 0, -
1): add_round_key(cipher_state,self._key_matrices[i]) inv_mix_columns(cipher_state)
inv_shift_rows(cipher_state) inv_sub_bytes(cipher_state)
add_round_key(cipher_state,self._key_matrices[0]) returnmatrix2bytes(cipher_state) def
encrypt_cbc(self,plaintext, iv): assert len(iv) ==16
```

```
plaintext = pad(plaintext)

blocks= [] previous= iv

forplaintext_block insplit_blocks(plaintext):

block= self.encrypt_block(xor_bytes(plaintext_block, previous)) blocks.append(block) previous
= block returnb''.join(blocks) def decrypt_cbc(self,ciphertext, iv): assert len(iv) ==16 blocks= []
previous= iv forciphertext_block in split_blocks(ciphertext):

blocks.append( xor_bytes(previous, self.decrypt_block(ciphertext_block))) previous =
ciphertext_block returnunpad(b''.join(blocks))

AES_KEY_SIZE = 16 SALT_SIZE= 16

defencrypt(key, plaintext, workload=100000): if isinstance(key, str): key= key.encode('utf-8') if
isinstance(plaintext, str): plaintext = plaintext.encode('utf-8')

salt = os.urandom(SALT_SIZE)

ciphertext = AES(key).encrypt_cbc(plaintext, key) returnciphertext

defdecrypt(key, ciphertext, workload=100000):

returnAES(key).decrypt_cbc(ciphertext, key)

if __name__=='__main__':

file1 = open("sample.txt","r+")

input_text = file1.read() file1.close() symmetric_key= input("Entersymmetrickey: ")
print(symmetric_key) cipher_text = encrypt(symmetric_key, input_text) file2 =
open("sample1.txt","w") file2.write(cipher_text)

 file2.close()

 file3 = open("sample1.txt","r+")

cipher_text_file= file3.read()

file3.close()

decrypt_text = decrypt(symmetric_key, cipher_text_file)

print('Decryped text:', decrypt_text)
```

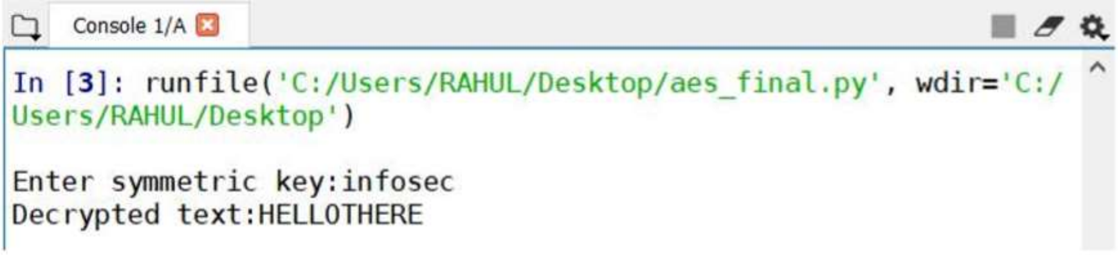
Output:

The text file “sample.txt” has the plain text.



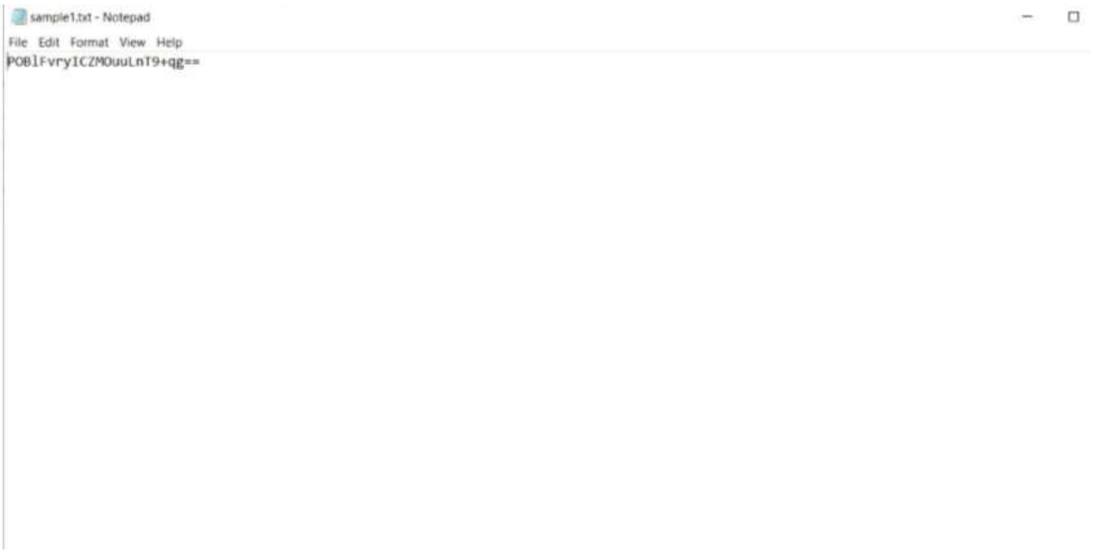
```
HELLOTHERE
```

The decrypted text is displayed.



```
In [3]: runfile('C:/Users/RAHUL/Desktop/aes_final.py', wdir='C:/
Users/RAHUL/Desktop')

Enter symmetric key:infosec
Decrypted text:HELLOTHERE
```

The text file “sample1.txt” has the encrypted file.



```
POBlFvryICZMOuuLnT9+qg==
```

-----------------------------------THANK YOU--