Part - A

1. With AES as the encryption / decryption algorithm, generate your own key.
2. Using ECB, CBC, CFB, OFB, and CTR modes to encrypt and decrypt your message.
3. Introduce errors in you plain text. Perform the encryption and decryption with the 5 operation modes again. Check how many blocks the errors propagated in each mode. Discuss if the propagations are as expected.

Answer:

The complete code is given below. It can be also found from below link (**https://github.com/TamjidHossain/CS654/blob/main/AES_5_Modes.py**).

The steps to answer questions 1, 2 and 3 are given below the code.

Code

```
1.  #!/usr/bin/env python3
2.  # -*- coding: utf-8 -*-
3.  """
4.  Created on Mon Nov  9 23:23:19 2020
5.
6.  @author: mdtamjidhossain
7.  """
8.
9.
10. #%%
11. # importing modules from library
12.
13. from Cryptodome.Cipher import AES
14.
15. import hashlib
16. import Padding
17. import binascii
18.
19. #%%
20. # Text color function
21. def colored(r, g, b, text):
22.     return "\033[38;2;{};{};{}m{} \033[38;2;255;255;255m".format(r, g, b, text)
23.
24. #%%
25. # encryption and decryption functions
26.
27. def encrypt(blockList_byte, key, mode):
28.     cipherList_byte = []
29.     encobj = AES.new(key,mode)
30.     for block in blockList_byte:
31.         cipherList_byte.append(encobj.encrypt(block))
32.
33.     return(cipherList_byte)
34.
35. def decrypt(cipherList_byte,key, mode):
36.     plainTextList_byte = []
37.     encobj = AES.new(key,mode)
38.     for block in cipherList_byte:
39.         plainTextList_byte.append(encobj.decrypt(block))
```

```
    return(plainTextList_byte)

def encrypt2(blockList_byte, key, mode, iv):
    cipherList_byte = []
    encobj = AES.new(key,mode, iv)
    for block in blockList_byte:
        cipherList_byte.append(encobj.encrypt(block))

    return(cipherList_byte)

def decrypt2(cipherList_byte,key, mode, iv):
    plainTextList_byte = []
    encobj = AES.new(key,mode,iv)
    for block in cipherList_byte:
        plainTextList_byte.append(encobj.decrypt(block))
    return(plainTextList_byte)

def encrypt3(blockList_byte, key, mode):
    cipherList_byte = []
    encobj = AES.new(key,AES.MODE_CTR)
    nonce = encobj.nonce
    for block in blockList_byte:
        cipherList_byte.append(encobj.encrypt(block))

    return(cipherList_byte, nonce)

def decrypt3(cipherList_byte,key, mode, nonce):
    plainTextList_byte = []
    encobj = AES.new(key,mode, nonce =  nonce)
    for block in cipherList_byte:
        plainTextList_byte.append(encobj.decrypt(block))
    return(plainTextList_byte)

#%%

def plainTextToByte(plaintext):
    plaintext = Padding.appendPadding(plaintext,blocksize=Padding.AES_blocksize,mode=0)
    plaintext_byte = plaintext.encode('utf-8')
    blockList_byte = [plaintext_byte[i:i+16] for i in range(0, len(plaintext_byte), 16)]
    return blockList_byte

def byteToHexBlock(blockList_byte):
    blockList_hex = []
    for byte in blockList_byte:
        blockList_hex.append(binascii.hexlify(byte).decode())
    return blockList_hex

#%%

# importing message from a file
plainTextLoc= '/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/plainText.txt'
# plainTextLoc= '/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/plainTextWithError.txt'
with open(plainTextLoc, 'r') as file:
    data = file.read().replace('\n', '')

# plaintext = data.encode('utf-8')
message = data

plaintext = message
#%%

#1. With AES as the encryption / decryption algorithm, generate your own key.

password ='cs654pass2020'
ival=12

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

```

```
iv= hex(ival)[2:8].zfill(16)


print(colored(255, 0, 0, 'AES Key (32 bytes or 256 bits):'))
print(str(key) + '\n')

print(colored(255, 0, 0, 'AES initialization vector (IV) :'))
print(str(iv) + '\n')

#%%
blockList_byte = []
blockList_hex = []

#%%
# encrypt and decrypt using ECB, CBC, CFB, OFB, CTR
print(colored(255, 0, 0, 'Original Message (string):'))
print(str(data) + '\n')

blockList_byte = plainTextToByte(data)
print(colored(255, 0, 0, 'Original Message after padding (bytes):'))
print(str(blockList_byte) + '\n')

blockList_hex = byteToHexBlock(blockList_byte)
print(colored(255, 0, 0, 'Original Message after padding (hex):'))
print(str(blockList_hex) + '\n')
print(colored(255, 0, 0, 'Encrypt and Decrypt using ECB, CBC, CFB, OFB, CTR :'))

#--------------------ECB starts----------------------------------------------
cipherList_byte = encrypt(blockList_byte,key, AES.MODE_ECB)
blockList_hex = byteToHexBlock(cipherList_byte)
print (colored(0, 255, 0, "  CipherBlock (ECB):\t")+str(blockList_hex))
print (colored(0, 255, 0, "  Ciphertext (ECB):\t")+ str(''.join(blockList_hex)))


plainTextList_byte = decrypt(cipherList_byte,key,AES.MODE_ECB)
plainTextList_str = []
for plaintext in plainTextList_byte:
    try:
        plainTextList_str.append(Padding.removePadding(plaintext.decode(),mode=0))
    except:
        plainTextList_str.append(plaintext.decode())
print (colored(0, 255, 0, "  PlaintextBlock (ECB):\t")+str(plainTextList_str))
print (colored(0, 255, 0, "  Plaintext (ECB):\t")+str(''.join(plainTextList_str))+ '\n')
#--------------------ECB ends----------------------------------------------


#--------------------CBC starts----------------------------------------------
cipherList_byte = encrypt2(blockList_byte,key, AES.MODE_CBC, iv.encode())
blockList_hex = byteToHexBlock(cipherList_byte)
print (colored(235, 204, 52, "  CipherBlock (CBC):\t  ")+str(blockList_hex))
print (colored(235, 204, 52, "  Ciphertext (CBC):\t  ")+ str(''.join(blockList_hex)))


plainTextList_byte = decrypt2(cipherList_byte,key,AES.MODE_CBC,iv.encode())
plainTextList_str = []
for plaintext in plainTextList_byte:
    try:
        plainTextList_str.append(Padding.removePadding(plaintext.decode(),mode=0))
    except:
        plainTextList_str.append(plaintext.decode())
print (colored(235, 204, 52, "  PlaintextBlock (CBC):  ")+str(plainTextList_str))
print (colored(235, 204, 52, "  Plaintext (CBC):\t  ")+str(''.join(plainTextList_str))+ '\n')
#--------------------CBC ends----------------------------------------------

#--------------------CFB starts----------------------------------------------
cipherList_byte = encrypt2(blockList_byte,key, AES.MODE_CFB, iv.encode())
blockList_hex = byteToHexBlock(cipherList_byte)
```

```
print (colored(52, 201, 235, "  CipherBlock (CFB):\t  ")+str(blockList_hex))
print (colored(52, 201, 235, "  Ciphertext (CFB):\t  ")+ str(''.join(blockList_hex)))


plainTextList_byte = decrypt2(cipherList_byte,key,AES.MODE_CFB,iv.encode())
plainTextList_str = []
for plaintext in plainTextList_byte:
    try:
        plainTextList_str.append(Padding.removePadding(plaintext.decode(),mode=0))
    except:
        plainTextList_str.append(plaintext.decode())
print (colored(52, 201, 235, "  PlaintextBlock (CFB):  ")+str(plainTextList_str))
print (colored(52, 201, 235, "  Plaintext (CFB):\t  ")+str(''.join(plainTextList_str)) + '\n')
#--------------------CFB ends-----------------------------------------

#--------------------OFB starts---------------------------------------
cipherList_byte = encrypt2(blockList_byte,key, AES.MODE_OFB, iv.encode())
blockList_hex = byteToHexBlock(cipherList_byte)
print (colored(183, 52, 235, "  CipherBlock (OFB):\t  ")+str(blockList_hex))
print (colored(183, 52, 235, "  Ciphertext (OFB):\t  ")+ str(''.join(blockList_hex)))


plainTextList_byte = decrypt2(cipherList_byte,key,AES.MODE_OFB,iv.encode())
plainTextList_str = []
for plaintext in plainTextList_byte:
    try:
        plainTextList_str.append(Padding.removePadding(plaintext.decode(),mode=0))
    except:
        plainTextList_str.append(plaintext.decode())
print (colored(183, 52, 235, "  PlaintextBlock (OFB):  ")+str(plainTextList_str))
print (colored(183, 52, 235, "  Plaintext (OFB):\t  ")+str(''.join(plainTextList_str))+ '\n')
#--------------------OFB ends-----------------------------------------


#--------------------CTR starts---------------------------------------
cipherList_byte, nonce = encrypt3(blockList_byte,key, AES.MODE_CTR)
blockList_hex = byteToHexBlock(cipherList_byte)
print (colored(235, 125, 52, "  CipherBlock (CTR):\t  ")+str(blockList_hex))
print (colored(235, 125, 52, "  Ciphertext (CTR):\t  ")+ str(''.join(blockList_hex)))


plainTextList_byte = decrypt3(cipherList_byte,key,AES.MODE_CTR,nonce)
plainTextList_str = []
for plaintext in plainTextList_byte:
    try:
        plainTextList_str.append(Padding.removePadding(plaintext.decode(),mode=0))
    except:
        plainTextList_str.append(plaintext.decode())
print (colored(235, 125, 52, "  PlaintextBlock (CTR):  ")+str(plainTextList_str))
print (colored(235, 125, 52, "  Plaintext (CTR):\t  ")+str(''.join(plainTextList_str))+ '\n')
nonce = ''
#--------------------CTR ends-----------------------------------------


#%%
```

Output

```
In [628]: runfile('/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/AES3.py', wdir='/Users/mdtamjidhossain/Fall-2020/Courses/CS654/
Project')
AES Key (32 bytes or 256 bits):
b'3\xa3\xa0\xd8\x87\xbb\x9e\x11\x8f\xe6\x95\xf2\xb6;\xffo}\xfdEm\xb9\xabe\xbb\x08\xb1R$ha\xdel'

AES initialization vector (IV) :
000000000000000c

Original Message (string):
CS654:Meet me at Sunday 7.00 PM near UNR quad.

Original Message after padding (bytes):
[b'CS654:Meet me at', b' Sunday 7.00 PM ', b'near UNR quad.\x02\x02']

Original Message after padding (hex):
['43533635343a4d656574206d65206174', '2053756e64617920372e303020504d20', '6e65617220554e5220717561642e0202']

Encrypt and Decrypt using ECB, CBC, CFB, OFB, CTR :
  CipherBlock (ECB):      ['860e79738a16fee2d2139629b806b769', 'e03eb9f67dc788779eae896ed8b90d34', '09d4e739b707282cefb88c5d8f90f790']
  Ciphertext (ECB):       860e79738a16fee2d2139629b806b769e03eb9f67dc788779eae896ed8b90d3409d4e739b707282cefb88c5d8f90f790
  PlaintextBlock (ECB):   ['CS654:Meet me at', ' Sunday 7.00 PM ', 'near UNR quad.']
  Plaintext (ECB):        CS654:Meet me at Sunday 7.00 PM near UNR quad.

  CipherBlock (CBC):      ['4d9353557f259fca5fd4b36c7d4b138b', '6091dd6d3c4dcc8382f8b31f92f41f3f', '74d17589f47c2bcfb912a1aa87b0b877']
  Ciphertext (CBC):       4d9353557f259fca5fd4b36c7d4b138b6091dd6d3c4dcc8382f8b31f92f41f3f74d17589f47c2bcfb912a1aa87b0b877
  PlaintextBlock (CBC):   ['CS654:Meet me at', ' Sunday 7.00 PM ', 'near UNR quad.']
  Plaintext (CBC):        CS654:Meet me at Sunday 7.00 PM near UNR quad.

  CipherBlock (CFB):      ['c44d118764d1d3f7f445146d954153f1', '0400e0d7ac54a56ce3f2b927a838ac07', 'b117ffed4b0bbfc6b6136e768ab34074']
  Ciphertext (CFB):       c44d118764d1d3f7f445146d954153f10400e0d7ac54a56ce3f2b927a838ac07b117ffed4b0bbfc6b6136e768ab34074
  PlaintextBlock (CFB):   ['CS654:Meet me at', ' Sunday 7.00 PM ', 'near UNR quad.']
  Plaintext (CFB):        CS654:Meet me at Sunday 7.00 PM near UNR quad.

  CipherBlock (OFB):      ['c4f7a27b6bd75902595ad20ed4fefb6e', 'e5bf6716309d54af8b1040bbd9e347e1', 'd8ab877852e5e540ffcbfb6a4c65ec37']
  Ciphertext (OFB):       c4f7a27b6bd75902595ad20ed4fefb6ee5bf6716309d54af8b1040bbd9e347e1d8ab877852e5e540ffcbfb6a4c65ec37
  PlaintextBlock (OFB):   ['CS654:Meet me at', ' Sunday 7.00 PM ', 'near UNR quad.']
  Plaintext (OFB):        CS654:Meet me at Sunday 7.00 PM near UNR quad.

  CipherBlock (CTR):      ['19384f9e1e69fd120f91890c307f39d8', '5f0df2d193c18a7ffd01e71b7c3683bd', '7ff7f68c07faff3cc2dbd2447d962066']
  Ciphertext (CTR):       19384f9e1e69fd120f91890c307f39d85f0df2d193c18a7ffd01e71b7c3683bd7ff7f68c07faff3cc2dbd2447d962066
  PlaintextBlock (CTR):   ['CS654:Meet me at', ' Sunday 7.00 PM ', 'near UNR quad.']
  Plaintext (CTR):        CS654:Meet me at Sunday 7.00 PM near UNR quad.
```

Fig. 1: Output of Part-A

1. With AES as the encryption / decryption algorithm, generate your own key.

Keys that are used in AES must be 128, 192, or 256 bits in size (for *AES-128*, *AES-192* or *AES-256* respectively). PyCryptodome supplies a function at Crypto.Random.get_random_bytes that returns a random byte string of a length we decide. To use this, we need to import the function and pass a length to the function. After running the code, we will get output key as shown in the below code snippet (Fig. 2).

```
In [569]: from Crypto.Random import get_random_bytes
     ...: key = get_random_bytes(32) # 32 bytes * 8 = 256 bits
     ...: print(colored(255,0,0,"key (32 bytes):\t")+ str(key))
key (32 bytes):    b'~\x8fX\x9a-\x0f\xb0\x95\xa4\x93x\xbd\xab\xb7\x9d%\xb2?\xd2\x11\xe0\xb0\x8d/0\xf3\xdc\x16\xbdT@-'
```

Fig. 2: Random AES key generation

We can also generate 32-bytes key using SHA-256 or PBKDF2 algorithm. For simplicity, in this project, SHA-256 has been used to generate a 32-bytes (256-bits) key using a password (see Fig. 3). This password can be –

- Produce from **User input**
- Generate from **Random function**
- Used as **Hard-coded** (not recommended)

In this project, hard-coded password has been used only to maintain consistency and find out the changes of ciphertext blocks due to an error against a predefined message.

Here, ***ival*** has been taken as any random number to generate an initialization vector for CBC, CFB, OFB mode. Initialization vector is just an arbitrary constant which is included in the hash function specification and is used as the initial hash value before any data is fed in.

```
#%%

#1. With AES as the encryption / decryption algorithm, generate your own key.

password ='cs654pass2020'
ival=12

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

iv= hex(ival)[2:8].zfill(16)


print(colored(255, 0, 0, 'AES Key (32 bytes or 256 bits):'))
print(str(key) + '\n')

print(colored(255, 0, 0, 'AES initialization vector (IV) :'))
print(str(iv) + '\n')
```

```
AES Key (32 bytes or 256 bits):
b'3\xa3\xa0\xd8\x87\xbb\x9e\x11\x8f\xe6\x95\xf2\xb6;\xffo}\xfdEm\xb9\xabe\xbb\x08\xb1R$ha\xdel'

AES initialization vector (IV) :
000000000000000c
```

Fig. 3: AES key generation using SHA-256

2. Using ECB, CBC, CFB, OFB, and CTR modes to encrypt and decrypt your message.

In this project,

Original message = ‘CS654:Meet me at Sunday 7.00 PM near UNR quad.
Manipulated message = ‘CS654:Meet me at Sunday 8.00 AM near UNR quad.

Instead of hard-coded the message, two files (*plainText.txt, plainTextWithError.txt*) are used to store the original and manipulated message accordingly. During each run, the code just simply reads the messages from one of the files. Fig.4 shows the code is reading original message from *plaintext.txt* file whereas *plainTextWithError.txt* has been commented out for this cycle.

```
#%%
# importing message from a file

plainTextLoc= '/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/plainText.txt'
# plainTextLoc= '/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/plainTextWithError.txt'
with open(plainTextLoc, 'r') as file:
    data = file.read().replace('\n', '')

message = data

plaintext = message
```

Fig. 4: Reading message from file

Below functions are used for encryption and decryption of the message.

- *'encrypt'* and *'decrypt'* functions are used for ECB mode where initialization vector or nonce are not required.
- *'encrypt2'* and *'decrypt2'* functions are used for CBC, CFB and OFB mode where initialization vector is required.
- *'encrypt3'* and *'decrypt3'* functions are used for CTR mode where nonce is required.

```
#%%
# encryption and decryption functions

def encrypt(blockList_byte, key, mode):
    cipherList_byte = []
    encobj = AES.new(key,mode)
    for block in blockList_byte:
        cipherList_byte.append(encobj.encrypt(block))

    return(cipherList_byte)

def decrypt(cipherList_byte,key, mode):
    plainTextList_byte = []
    encobj = AES.new(key,mode)
    for block in cipherList_byte:
        plainTextList_byte.append(encobj.decrypt(block))
    return(plainTextList_byte)

def encrypt2(blockList_byte, key, mode, iv):
    cipherList_byte = []
    encobj = AES.new(key,mode, iv)
    for block in blockList_byte:
        cipherList_byte.append(encobj.encrypt(block))

    return(cipherList_byte)

def decrypt2(cipherList_byte,key, mode, iv):
    plainTextList_byte = []
    encobj = AES.new(key,mode,iv)
    for block in cipherList_byte:
        plainTextList_byte.append(encobj.decrypt(block))
    return(plainTextList_byte)

def encrypt3(blockList_byte, key, mode):
    cipherList_byte = []
    encobj = AES.new(key,AES.MODE_CTR)
    nonce = encobj.nonce
    for block in blockList_byte:
        cipherList_byte.append(encobj.encrypt(block))

    return(cipherList_byte, nonce)

def decrypt3(cipherList_byte,key, mode, nonce):
    plainTextList_byte = []
    encobj = AES.new(key,mode, nonce =  nonce)
    for block in cipherList_byte:
        plainTextList_byte.append(encobj.decrypt(block))
    return(plainTextList_byte)

```

Fig. 5: Encryption and Decryption function

There are two supplemental functions as well (see figure 6). *'plainTextToByte(plaintext)'* is used to convert the plaintext into bytes and it also separates the bytes into multiple 16-byte blocks (AES block size is 16-bytes long). However, before the start of conversion, plaintext is padded to make it multiple of AES block size (multiple of 16-bytes, i.e., 16-bytes or 32 bytes or 64 bytes and so on).

'byteToHexBlock(blockList_byte)' converts the blocks of byte s into blocks of hex.

```
#%%

def plainTextToByte(plaintext):
    plaintext = Padding.appendPadding(plaintext,blocksize=Padding.AES_blocksize,mode=0)
    plaintext_byte = plaintext.encode('utf-8')
    blockList_byte = [plaintext_byte[i:i+16] for i in range(0, len(plaintext_byte), 16)]
    return blockList_byte

def byteToHexBlock(blockList_byte):
    blockList_hex = []
    for byte in blockList_byte:
        blockList_hex.append(binascii.hexlify(byte).decode())
    return blockList_hex

```

Fig. 6: supplemental functions (*plainTextToByte(plaintext), byteToHexBlock(blockList_byte))*

First, the original message, the padded blocks of bytes and the padded blocks of hex of the message are printed using below code (see figure 7).

```
#%%
# encrypt and decrypt using ECB, CBC, CFB, OFB, CTR
print(colored(255, 0, 0, 'Original Message (string):'))
print(str(data) + '\n')

blockList_byte = plainTextToByte(data)
print(colored(255, 0, 0, 'Original Message after padding (bytes):'))
print(str(blockList_byte) + '\n')

blockList_hex = byteToHexBlock(blockList_byte)
print(colored(255, 0, 0, 'Original Message after padding (hex):'))
print(str(blockList_hex) + '\n')
print(colored(255, 0, 0, 'Encrypt and Decrypt using ECB, CBC, CFB, OFB, CTR :'))

```

```
Original Message (string):
CS654:Meet me at Sunday 7.00 PM near UNR quad.

Original Message after padding (bytes):
[b'CS654:Meet me at', b' Sunday 7.00 PM ', b'near UNR quad.\x02\x02']

Original Message after padding (hex):
['43533635343a4d656574206d65206174', '2053756e64617920372e303020504d20', '6e65617220554e5220717561642e0202']
```

Fig. 7: Original Message (string, byte and hex formats)

In every mode, the code follows some basic steps. These are–

- Encrypting blocks of bytes
- Convert block of bytes into block of hex for better representation
- Decrypting blocks of bytes
- Removing padding
- Printing the ciphertext and plaintext

Apart from these basic steps, CBC, CFB and OFB mode takes initialization vector as an added argument during encryption and decryption. However, CTR mode takes nonce (a number used only once) while ECB modes neither takes initialization vector nor nonce.

ECB Mode:

Code:

```
#--------------------ECB starts----------------------------------------------------
cipherList_byte = encrypt(blockList_byte,key, AES.MODE_ECB)
blockList_hex = byteToHexBlock(cipherList_byte)
print (colored(0, 255, 0, "  CipherBlock (ECB):\t")+str(blockList_hex))
print (colored(0, 255, 0, "  Ciphertext (ECB):\t")+ str(''.join(blockList_hex)))


plainTextList_byte = decrypt(cipherList_byte,key,AES.MODE_ECB)
plainTextList_str = []
for plaintext in plainTextList_byte:
    try:
        plainTextList_str.append(Padding.removePadding(plaintext.decode(),mode=0))
    except:
        plainTextList_str.append(plaintext.decode())
print (colored(0, 255, 0, "  PlaintextBlock (ECB):\t")+str(plainTextList_str))
print (colored(0, 255, 0, "  Plaintext (ECB):\t")+str(''.join(plainTextList_str))+ '\n')
#--------------------ECB ends------------------------------------------------------
```

Fig. 9: Code of ECB Mode

Output:

```
Encrypt and Decrypt using ECB, CBC, CFB, OFB, CTR :
  CipherBlock (ECB):      ['860e79738a16fee2d2139629b806b769', 'e03eb9f67dc788779eae896ed8b90d34', '09d4e739b707282cefb88c5d8f90f790']
  Ciphertext (ECB):       860e79738a16fee2d2139629b806b769e03eb9f67dc788779eae896ed8b90d3409d4e739b707282cefb88c5d8f90f790
  PlaintextBlock (ECB):   ['CS654:Meet me at', ' Sunday 7.00 PM ', 'near UNR quad.']
  Plaintext (ECB):        CS654:Meet me at Sunday 7.00 PM near UNR quad.
```

Fig. 10: Output of ECB Mode

CBC Mode:

Code:

```
#--------------------CBC starts----------------------------------------------------
cipherList_byte = encrypt2(blockList_byte,key, AES.MODE_CBC, iv.encode())
blockList_hex = byteToHexBlock(cipherList_byte)
print (colored(235, 204, 52, "  CipherBlock (CBC):\t   ")+str(blockList_hex))
print (colored(235, 204, 52, "  Ciphertext (CBC):\t   ")+ str(''.join(blockList_hex)))


plainTextList_byte = decrypt2(cipherList_byte,key,AES.MODE_CBC,iv.encode())
plainTextList_str = []
for plaintext in plainTextList_byte:
    try:
        plainTextList_str.append(Padding.removePadding(plaintext.decode(),mode=0))
    except:
        plainTextList_str.append(plaintext.decode())
print (colored(235, 204, 52, "  PlaintextBlock (CBC):  ")+str(plainTextList_str))
print (colored(235, 204, 52, "  Plaintext (CBC):\t   ")+str(''.join(plainTextList_str))+ '\n')
#--------------------CBC ends------------------------------------------------------
```

Fig. 11: Code of CBC Mode

Output:

```
  CipherBlock (CBC):      ['4d9353557f259fca5fd4b36c7d4b138b', '6091dd6d3c4dcc8382f8b31f92f41f3f', '74d17589f47c2bcfb912a1aa87b0b877']
  Ciphertext (CBC):       4d9353557f259fca5fd4b36c7d4b138b6091dd6d3c4dcc8382f8b31f92f41f3f74d17589f47c2bcfb912a1aa87b0b877
  PlaintextBlock (CBC):   ['CS654:Meet me at', ' Sunday 7.00 PM ', 'near UNR quad.']
  Plaintext (CBC):        CS654:Meet me at Sunday 7.00 PM near UNR quad.
```

Fig. 12: Output of CBC Mode

CFB Mode:

Code:

```
#--------------------CFB starts-----------------------------------------------------
cipherList_byte = encrypt2(blockList_byte,key, AES.MODE_CFB, iv.encode())
blockList_hex = byteToHexBlock(cipherList_byte)
print (colored(52, 201, 235, "  CipherBlock (CFB):\t   ")+str(blockList_hex))
print (colored(52, 201, 235, "  Ciphertext (CFB):\t   ")+ str(''.join(blockList_hex)))

plainTextList_byte = decrypt2(cipherList_byte,key,AES.MODE_CFB,iv.encode())
plainTextList_str = []
for plaintext in plainTextList_byte:
    try:
        plainTextList_str.append(Padding.removePadding(plaintext.decode(),mode=0))
    except:
        plainTextList_str.append(plaintext.decode())
print (colored(52, 201, 235, "  PlaintextBlock (CFB):  ")+str(plainTextList_str))
print (colored(52, 201, 235, "  Plaintext (CFB):\t   ")+str(''.join(plainTextList_str)) + '\n')
#--------------------CFB ends-------------------------------------------------------
```

Fig. 13: Code of CFB Mode

Output:

```
CipherBlock (CFB):      ['c44d118764d1d3f7f445146d954153f1', '0400e0d7ac54a56ce3f2b927a838ac07', 'b117ffed4b0bbfc6b6136e768ab34074']
Ciphertext (CFB):       c44d118764d1d3f7f445146d954153f10400e0d7ac54a56ce3f2b927a838ac07b117ffed4b0bbfc6b6136e768ab34074
PlaintextBlock (CFB):   ['CS654:Meet me at', ' Sunday 7.00 PM ', 'near UNR quad.']
Plaintext (CFB):        CS654:Meet me at Sunday 7.00 PM near UNR quad.
```

Fig. 14: Output of CFB Mode

OFB Mode:

Code:

```
#--------------------OFB starts-----------------------------------------------------
cipherList_byte = encrypt2(blockList_byte,key, AES.MODE_OFB, iv.encode())
blockList_hex = byteToHexBlock(cipherList_byte)
print (colored(183, 52, 235, "  CipherBlock (OFB):\t   ")+str(blockList_hex))
print (colored(183, 52, 235, "  Ciphertext (OFB):\t   ")+ str(''.join(blockList_hex)))


plainTextList_byte = decrypt2(cipherList_byte,key,AES.MODE_OFB,iv.encode())
plainTextList_str = []
for plaintext in plainTextList_byte:
    try:
        plainTextList_str.append(Padding.removePadding(plaintext.decode(),mode=0))
    except:
        plainTextList_str.append(plaintext.decode())
print (colored(183, 52, 235, "  PlaintextBlock (OFB):  ")+str(plainTextList_str))
print (colored(183, 52, 235, "  Plaintext (OFB):\t   ")+str(''.join(plainTextList_str))+ '\n')
#--------------------OFB ends-------------------------------------------------------
```

Fig. 15: Code of OFB Mode

Output:

```
CipherBlock (OFB):      ['c4f7a27b6bd75902595ad20ed4fefb6e', 'e5bf6716309d54af8b1040bbd9e347e1', 'd8ab877852e5e540ffcbfb6a4c65ec37']
Ciphertext (OFB):       c4f7a27b6bd75902595ad20ed4fefb6ee5bf6716309d54af8b1040bbd9e347e1d8ab877852e5e540ffcbfb6a4c65ec37
PlaintextBlock (OFB):   ['CS654:Meet me at', ' Sunday 7.00 PM ', 'near UNR quad.']
Plaintext (OFB):        CS654:Meet me at Sunday 7.00 PM near UNR quad.
```

Fig. 16: Output of OFB Mode

CTR Mode:

Code:

```
#--------------------CTR starts-----------------------------------------------
cipherList_byte, nonce = encrypt3(blockList_byte,key, AES.MODE_CTR)
blockList_hex = byteToHexBlock(cipherList_byte)
print (colored(235, 125, 52, "  CipherBlock (CTR):\t   ")+str(blockList_hex))
print (colored(235, 125, 52, "  Ciphertext (CTR):\t   ")+ str(''.join(blockList_hex)))


plainTextList_byte = decrypt3(cipherList_byte,key,AES.MODE_CTR,nonce)
plainTextList_str = []
for plaintext in plainTextList_byte:
    try:
        plainTextList_str.append(Padding.removePadding(plaintext.decode(),mode=0))
    except:
        plainTextList_str.append(plaintext.decode())
print (colored(235, 125, 52, "  PlaintextBlock (CTR):  ")+str(plainTextList_str))
print (colored(235, 125, 52, "  Plaintext (CTR):\t   ")+str(''.join(plainTextList_str))+ '\n')
nonce = ''
#--------------------CTR ends-------------------------------------------------
```

Fig. 15: Code of CTR Mode

Output:

```
CipherBlock (CTR):      ['19384f9e1e69fd120f91890c307f39d8', '5f0df2d193c18a7ffd01e71b7c3683bd', '7ff7f68c07faff3cc2dbd2447d962066']
Ciphertext (CTR):       19384f9e1e69fd120f91890c307f39d85f0df2d193c18a7ffd01e71b7c3683bd7ff7f68c07faff3cc2dbd2447d962066
PlaintextBlock (CTR):   ['CS654:Meet me at', ' Sunday 7.00 PM ', 'near UNR quad.']
Plaintext (CTR):        CS654:Meet me at Sunday 7.00 PM near UNR quad.
```

Fig. 16: Output of CTR Mode

3. Introduce errors in you plain text. Perform the encryption and decryption with the 5 operation modes again. Check how many blocks the errors propagated in each mode. Discuss if the propagations are as expected.

Error message: 'CS654:Meet me at Sunday 8.00 AM near UNR quad.'
Output:

```
In [629]: runfile('/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/AES3.py', wdir='/Users/mdtamjidhossain/Fall-2020/Courses/CS654/
Project')
AES Key (32 bytes or 256 bits):
b'3\xa3\xa0\xd8\x87\xbb\x9e\x11\x8f\xe6\x95\xf2\xb6;\xffo}\xfdEm\xb9\xabe\xbb\x08\xb1R$ha\xdel'

AES initialization vector (IV) :
000000000000000c

Original Message (string):
CS654:Meet me at Sunday 8.00 AM near UNR quad.

Original Message after padding (bytes):
[b'CS654:Meet me at', b' Sunday 8.00 AM ', b'near UNR quad.\x02\x02']

Original Message after padding (hex):
['43533635343a4d656574206d65206174', '2053756e64617920382e303020414d20', '6e65617220554e5220717561642e0202']

Encrypt and Decrypt using ECB, CBC, CFB, OFB, CTR :
  CipherBlock (ECB):      ['860e79738a16fee2d2139629b806b769', 'e328005c76bf22aa74c2cbfbac206488', '09d4e739b707282cefb88c5d8f90f790']
  Ciphertext (ECB):       860e79738a16fee2d2139629b806b769e328005c76bf22aa74c2cbfbac20648809d4e739b707282cefb88c5d8f90f790
  PlaintextBlock (ECB):   ['CS654:Meet me at', ' Sunday 8.00 AM ', 'near UNR quad.']
  Plaintext (ECB):        CS654:Meet me at Sunday 8.00 AM near UNR quad.

  CipherBlock (CBC):      ['4d9353557f259fca5fd4b36c7d4b138b', '397b77ee9bec0e08a399d1755908a530', '1f1809970ddef918f74498bfe5140b5b']
  Ciphertext (CBC):       4d9353557f259fca5fd4b36c7d4b138b397b77ee9bec0e08a399d1755908a5301f1809970ddef918f74498bfe5140b5b
  PlaintextBlock (CBC):   ['CS654:Meet me at', ' Sunday 8.00 AM ', 'near UNR quad.']
  Plaintext (CBC):        CS654:Meet me at Sunday 8.00 AM near UNR quad.

  CipherBlock (CFB):      ['c44d118764d1d3f7f445146d954153f1', '0400e0d7ac54a56cecd8112fd52129f1', '8b7716b7564a1345ead7ead02196deaf']
  Ciphertext (CFB):       c44d118764d1d3f7f445146d954153f10400e0d7ac54a56cecd8112fd52129f18b7716b7564a1345ead7ead02196deaf
  PlaintextBlock (CFB):   ['CS654:Meet me at', ' Sunday 8.00 AM ', 'near UNR quad.']
  Plaintext (CFB):        CS654:Meet me at Sunday 8.00 AM near UNR quad.

  CipherBlock (OFB):      ['c4f7a27b6bd75902595ad20ed4fefb6e', 'e5bf6716309d54af841040bbd9f247e1', 'd8ab877852e5e540ffcbfb6a4c65ec37']
  Ciphertext (OFB):       c4f7a27b6bd75902595ad20ed4fefb6ee5bf6716309d54af841040bbd9f247e1d8ab877852e5e540ffcbfb6a4c65ec37
  PlaintextBlock (OFB):   ['CS654:Meet me at', ' Sunday 8.00 AM ', 'near UNR quad.']
  Plaintext (OFB):        CS654:Meet me at Sunday 8.00 AM near UNR quad.

  CipherBlock (CTR):      ['3c4235758240345bf43af4ff07e91f73', 'cbc08f852ec7f195f9f74145e295801d', '39edb75b6901b756f1f888f142d99eb3']
  Ciphertext (CTR):       3c4235758240345bf43af4ff07e91f73cbc08f852ec7f195f9f74145e295801d39edb75b6901b756f1f888f142d99eb3
  PlaintextBlock (CTR):   ['CS654:Meet me at', ' Sunday 8.00 AM ', 'near UNR quad.']
  Plaintext (CTR):        CS654:Meet me at Sunday 8.00 AM near UNR quad.
```

Figure 17: Output with error message

We have to read message from *'plainTextWithError.txt'*. For simplicity and consistency, AES key and initialization vector has been kept same. After running the code again, it can be seen that, the error has been reflected in all the formats of the message (string/byte/hex). The changes are shown by yellow boxes in fig. 18.

```
Original Message (string):
CS654:Meet me at Sunday 7.00 PM near UNR quad.

Original Message after padding (bytes):
[b'CS654:Meet me at', b' Sunday 7.00 PM ', b'near UNR quad.\x02\x02']

Original Message after padding (hex):
['43533635343a4d656574206d65206174', '2053756e64617920372e303020504d20', '6e65617220554e5220717561642e0202']
```

Original message

```
Original Message (string):
CS654:Meet me at Sunday 8.00 AM near UNR quad.

Original Message after padding (bytes):
[b'CS654:Meet me at', b' Sunday 8.00 AM ', b'near UNR quad.\x02\x02']

Original Message after padding (hex):
['43533635343a4d656574206d65206174', '2053756e64617920382e303020414d20', '6e65617220554e5220717561642e0202']
```

Message with error

Figure 18: Original message vs error message

After analyzing output with original message (figure 1) and output with error messaged (figure 17), below observations can be pointed out –

1. Error doesn't propagate in ECB. We know, in ECB, each block is encrypted independently. Our observation also suggests so. From below cipher blocks, it can be seen that, only the second (2nd) block is affected due to the errors in the 2nd block.1st and 3rd blocks are same for both original text and error text.

CipherBlock (ECB)_original text: ['860e79738a16fee2d2139629b806b769', 'e03eb9f67dc788779eae896ed8b90d34', '09d4e739b707282cefb88c5d8f90f790']

CipherBlock (ECB) _error text: ['860e79738a16fee2d2139629b806b769', 'e328005c76bf22aa74c2cbfbac206488', '09d4e739b707282cefb88c5d8f90f790']

2. Error propagates to subsequent blocks in CBC. However, the error also changes the entire ciphertext block where it takes place. From below cipher blocks, it can be seen that due to the errors in the 2nd block, both 2nd block and the 3rd block are affected and changed entirely.

CipherBlock (CBC)_original text: ['4d9353557f259fca5fd4b36c7d4b138b', '6091dd6d3c4dcc8382f8b31f92f41f3f', '74d17589f47c2bcfb912a1aa87b0b877']

CipherBlock (CBC) _error text: ['4d9353557f259fca5fd4b36c7d4b138b', '397b77ee9bec0e08a399d1755908a530', '1f1809970ddef918f74498bfe5140b5b']

3. Error propagates to subsequent blocks in CFB. However, the error doesn't change the entire ciphertext block where it takes place; rather the block changes from the bit position where the error first occurs. From below cipher blocks, it can be seen that due to the errors in the 2nd block, both 2nd block and the 3rd block are affected and changed.

CipherBlock (CFB) _original text: ['c44d118764d1d3f7f445146d954153f1', '0400e0d7ac54a56ce3f2b927a838ac07', 'b117ffed4b0bbfc6b6136e768ab34074']

CipherBlock (CFB) _error text: ['c44d118764d1d3f7f445146d954153f1', '0400e0d7ac54a56cecd8112fd52129f1', '8b7716b7564a1345ead7ead02196deaf']

4. Error doesn't propagate to subsequent blocks. The bit error in plaintext may only affect the corresponding bit of ciphertext. From below cipher blocks, it can be seen that due to the bit errors in the 2nd block, only those bits are affected in ciphertext

CipherBlock (OFB) _original text: ['c4f7a27b6bd75902595ad20ed4fefb6e', 'e5bf6716309d54af8b1040bbd9e347e1', 'd8ab877852e5e540ffcbfb6a4c65ec37']

CipherBlock (OFB) _error text: ['c4f7a27b6bd75902595ad20ed4fefb6e', 'e5bf6716309d54af841040bbd9f247e1', 'd8ab877852e5e540ffcbfb6a4c65ec37']

5. Error doesn't propagate to subsequent blocks. However, as nonce is random and changes during each iteration with the same key, the ciphertexts are different with original text and error text. Even, for same text, the ciphertext would be different for each run of the code.

CipherBlock (CTR) _original text: ['624ce7fd4b5995162aa7bbda8fc3f9d7', 'e5e2493da573ea00ebbec98a223f8f0e', '7413b82cea8c7f4afa92e7e84a655c3b']

CipherBlock (CTR) _error text : ['1829efd489543fd59cae128547b13de8', '2d05bd806befad2602c5f7ee419be429', '087c674c18c36cba9bd51ea987a55c91']

So, it can be told that, in this project, the error in plaintext follows the mechanisms of each different modes.

PART -2

Use RSA to encrypt and decrypt the same message (you can follow the steps shown in textbook Figure 9.7). Compare the time consumption with AES. (You may need to make the message long enough to see the significant difference in time.)

Answer

The code for encrypting and decrypting below message using RSA and AES encryption technique can be found from below link.

Message:
(source: *Shen, H. and Domenic Forte. "Nanopyramid: An Optical Scrambler Against Backside Probing Attacks." (2018)*

Optical probing from the backside of an integrated circuit (IC) is a powerful failure analysis technique but raises serious security concerns when in the hands of attackers. For instance, attacks using laser voltage probing (LVP) allow direct reading of sensitive information being stored and/or processed in the IC.

Code Link:
RSA: https://github.com/TamjidHossain/CS654/blob/main/RSA.py
AES: https://github.com/TamjidHossain/CS654/blob/main/AES_CTR.py

RSA Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Nov 11 12:04:42 2020

@author: mdtamjidhossain
"""
#%%
from Cryptodome.PublicKey import RSA
from Cryptodome.Cipher import PKCS1_OAEP
import binascii

import time
```

```
#%%

start = time.time()
#%%

# Text color function
def colored(r, g, b, text):
    return "\033[38;2;{};{};{}m{} \033[38;2;255;255;255m".format(r, g, b, text)

#%%
# importing message from a file

plainTextLoc= '/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/plainText_long.txt'
# plainTextLoc= '/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/plainTextWithError.txt'
with open(plainTextLoc, 'r') as file:
    data = file.read().replace('\n', '')

message = data

plaintext = message
#%%

keyPair = RSA.generate(3072)

pubKey = keyPair.publickey()
print(colored(255, 0, 0, "Public key:"))
print(f"(n={hex(pubKey.n)}, e={hex(pubKey.e)})" + '\n')

pubKeyPEM = pubKey.exportKey()
print(pubKeyPEM.decode('ascii')+ '\n')

print(colored(255, 0, 0, "Private key:"))
print(f"(n={hex(pubKey.n)}, d={hex(keyPair.d)})"+'\n')
privKeyPEM = keyPair.exportKey()
print(privKeyPEM.decode('ascii')+'\n\n')
#%%
# Encryption

encryptor = PKCS1_OAEP.new(pubKey)
encrypted = encryptor.encrypt(plaintext.encode())
print(colored(255,0,0, "Encrypted:\n"), binascii.hexlify(encrypted), '\n')
#%%
# Decryption

decryptor = PKCS1_OAEP.new(keyPair)
decrypted = decryptor.decrypt(encrypted)
print(colored(255,0,0,'Decrypted:\n'), decrypted, '\n\n')
#%%

end = time.time()
print(colored(255,0,0, 'Execution TIme: '), end - start)
#%%
```

Output

```
In [732]: runfile('/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/RSA.py', wdir='/Users/mdtamjidhossain/Fall-2020/Courses/CS654/
Project')
Public key:
(n=0xda4e0a55270fa51c737b8d8a9bb877b1afcd84f09960eedbd8ee044a4fa64549ae0c1c16f56c6bc5af64e7da3e3a6f2237304468e03425a2265c1066e2f105ba340cc76f7
e5d4ea7372b43b58828dee66a45191ccbff9ca1e6ccae2a8dfc90c066e7500e909ccdb401d2490fc49b8d02707e6a5bbd1f9214d19f420aec358c12373c1adfc67c6db5678d5f3
cc3adf0cfdb90981a8599c5b02c47be64f3a9ebab6b3aeb084d633f8820489809711cee400532a4a8170e5ef0446e7fe5ae20f5f00b169a443d0933089acd22886e57131448c02
c8cf1c76d3238f7bd11eaabfe4eac4fc8ae17c79d8a8e96013eb5e8552c0cae22bf68f4b75ccea054df8d0c45f42fc64d31f789eb4b3fa767f0fe6fc39b6b6c93b6c06f7a7743
7c588a666e30efb051378e4c4bbcbd9c92a47365086e74ace078b2b82f237ec7811047ea42b0cf4ecf2958ddd1c36120b05d2e524218a84f3b3340f640cddc54537c9a0c967ed1
f8e47ec2e11bb9f9efe087706bf788a0c55a6e39a950127067515, e=0x10001)

-----BEGIN PUBLIC KEY-----
MIIBojANBgkqhkiG9w0BAQEFAAOCAY8AMIIBigKCAYEA2k4KVScPpRxze42Km7h3
sa/NhPCZYO7b2O4ESk+mRUmuDBwW9Wxrxa9k59o+Om8iNzBEaOA0JaImXBBm4vEF
ujQMx297Ll1OpzcrQ7WIKN7makUZHMv/nKHmzK4qjfyQwGbnUA6QnM20AdJJD8Sb
jQJwfmpbvR+SFNGfQgrsNYwSNzwa38Z8bbVnjV86DMOt8M/bkJgahZnFsCxHvmTz
qeurazrrCE1jP4ggSJgJcRzuQAUypKgXDl7wRG5/5a4g9fALFppEPQkzCJrNIohu
VxMUSMAsHIzxx20yOPe9Eeqr/k6sT8iuF8edio6WAT616FUsDK4iv2j0t1zOoFTf
jQxF9C/GTTH3ietLP6dn8P5vw5trbJO2wG96d0NOB8WIpmbjDvsFE3jkxLvL2ckq
RzZQhudKzgeLK4LyN+x4EQR+pCsM9OzylY3dHDYSCwXS5SQhioTzszQPZAzdxUU3
yaDJZ+0Yv45H7C4Ru5+e/gh3Br94igxVpuOalQEnBnUVAgMBAAE=
-----END PUBLIC KEY-----

Private key:
(n=0xda4e0a55270fa51c737b8d8a9bb877b1afcd84f09960eedbd8ee044a4fa64549ae0c1c16f56c6bc5af64e7da3e3a6f2237304468e03425a2265c1066e2f105ba340cc76f7
e5d4ea7372b43b58828dee66a45191ccbff9ca1e6ccae2a8dfc90c066e7500e909ccdb401d2490fc49b8d02707e6a5bbd1f9214d19f420aec358c12373c1adfc67c6db5678d5f3
cc3adf0cfdb90981a8599c5b02c47be64f3a9ebab6b3aeb084d633f8820489809711cee400532a4a8170e5ef0446e7fe5ae20f5f00b169a443d0933089acd22886e57131448c02
c8cf1c76d3238f7bd11eaabfe4eac4fc8ae17c79d8a8e96013eb5e8552c0cae22bf68f4b75ccea054df8d0c45f42fc64d31f789eb4b3fa767f0fe6fc39b6b6c93b6c06f7a7743
7c588a666e30efb051378e4c4bbcbd9c92a47365086e74ace078b2b82f237ec7811047ea42b0cf4ecf2958ddd1c36120b05d2e524218a84f3b3340f640cddc54537c9a0c967ed1
f8e47ec2e11bb9f9efe087706bf788a0c55a6e39a950127067515,
d=0x105a07f0ef65ff04807643789ba7e7ad48a1660bb338126ed212fd56c74709746bf943bc5e2c92c3acb02bf95f5a61fb27de46c2d7f25d5445e274fc63e8d0f8977aa7e5f1
dc4183df15c4f2aac22c4a098691f63925dfbe0a75e53627a05603acb6e1a5544f2e78452a52bc7fdf4a8d6179929965b33643b6c78d5098a1fd2885cdcbd1dc5748776df6d0a8
8ecfaf5493c4c2f93a43204e8d23e03ec5b7ecab65c834bfd088cbd8342f83d5dc8cb7202bfe2a5d0881becf93eca14a0604161823ef626a255aea4e53d908cacfd094c5c5855a
525aded73289e693b6a3ce56dc2ab2bee7b77c52fddc2720b599bf38e8f85db2673d0947752ea2bc3e128a95d10d0cea880160bf9a226cf0d99ac703899407cbb2e3f5e9c857d4
25a5d491c7125db154d917c297527f5de571c93163625eaaec3eec734fa5b92e2ff3e8d1f3ff6702e6960087e5c84fe952dcba55ad3965d57de6dee86ab9e8736cea3b4cbf55af
7c311fb19211d18cc99981208e33bbf44d72223fd0a0a7a5db99)

-----BEGIN RSA PRIVATE KEY-----
MIIG5AIBAAKCAYEA2k4KVScPpRxze42Km7h3sa/NhPCZYO7b2O4ESk+mRUmuDBwW
9Wxrxa9k59o+Om8iNzBEaOA0JaImXBBm4vEFujQMx297Ll1OpzcrQ7WIKN7makUZ
HMv/nKHmzK4qjfyQwGbnUA6QnM20AdJJD8SbjQJwfmpbvR+SFNGfQgrsNYwSNzwa
38Z8bbVnjV86DMOt8M/bkJgahZnFsCxHvmTzqeurazrrCE1jP4ggSJgJcRzuQAUy
pKgXDl7wRG5/5a4g9fALFppEPQkzCJrNIohuVxMUSMAsHIzxx20yOPe9Eeqr/k6s
T8iuF8edio6WAT616FUsDK4iv2j0t1zOoFTfjQxF9C/GTTH3ietLP6dn8P5vw5tr
bJO2wG96d0NOB8WIpmbjDvsFE3jkxLvL2ckqRzZQhudKzgeLK4LyN+x4EQR+pCsM
9OzylY3dHDYSCwXS5SQhioTzszQPZAzdxUU3yaDJZ+0Yv45H7C4Ru5+e/gh3Br94
igxVpuOalQEnBnUVAgMBAAECggGAEFoH8O9l/wSAdkN4m6fnrUihZguzOBJu0hL9
VsdHCXRr+UO8XiySw6ywK/lfWmH7J95GwtfyXVRF4nT8Y+jQ+Jd6p+XxjdxBg98V
xPKqwixKCYaR9jkl374KdeU2J6BWA6y24aVUTy54RSpSvH/fSo1heZKZZbM2Q7bH
jVCYof0ohc3L0dxXSHdt9tCoc47Pr1STxML5OkMgTo0j4D7Ft+yrZcg0v9CIy9g0
L4PV3Iy3ICv+Kl0Igb7Pk+yhSgYEFhgj72JqJVrqTlPZCMrP0JTFxYVa0VJa3tcy
ieaTtqPOVtwqsr7nt3xS/dwnILWZvzjo+F2yZz0JR3Uuorw+EoqV0Q0M6ogBYL+a
Imzw2ZrHA4mUB8uy4/XpyFfUgCWl1JHHEl2xVNkXwpdSf13lcckxY2Jequw+7HNP
pbkuL/Po0fP/ZwLmlgCH5chP6VLculWtOWXVfebe6Gq56HNs6jtMv1WvJnwxH7GS
EdGMyZmBII4zu/RNciI/0KCnpduZAoHBAOaRhdY3adIHhjGmo3AR+RI6li7d6on+
FmEml+kYC6A17/8RzYGwwa8cNfbEbUGnBrDh0xG125ypj9V7kriSN7rw/BvyMbMb
bQwvSAVPYbxhj1yBtMBGk45veOA2CadB+UevwoVO5Xw2gK0+Qhz8ExEmjtF86Hx5
kHSGclu5pmC/e5fV4GaPhto6mgPuTvTsjH1JTEPDxVVF3n1obiImW6yg7U7w8LZP
jxVOQEaGfRipGsk8LYGb01q2oWQY7t5MLQKBwQDyYjwTOBNY7dUWOJhg0R+8SDFJ
Rvh2R4Xv7BOhKBWc7bsmTVndwTnezxON1kjhRzgh88Y4X0F3wVpmh5riKjoOZrIx
r0p5oPcAhxUFzH3BzEU5AhGV9MSuuP7i+NUbhBz2mdZnRjq5X60fYelK4o9ro3VH
X9IrXjEfgj3hQeHPGkGjwn4JIccjbHA6NLwNUaRjGMVtdtn1YrMfM2EyCIQlOlsT
abCGiyn1a3HdEgImi3aJw9tpKI9qpsovKn8QFYkCgcBBCjXdyr3gEF1Lbj1WZMuE
19/7koARswovF4rspurIo85yflwhRHIual/Sn3BOf8ZEaT94FHvBwpcfWn9171En
O8lyPKzGYGsvb7r6PH1OfZrKNpB9WKtRF6LaZIPumY6vKSZPIoVbt6gutoxBzt22
DdFlyL1wN36HUFG8DC+hxQIhP6TQaZZZwwQlLbk6H+w1f1KzHutwdYggii/fFEDx
/mdSyGaulty+3tcEzfVF2pYCxCX2FLXXNgNq0Bm5vXUCgcEA2IkSCklqX2P5naRp
3PR2WYpVHZWdmm1DJpfM4Ygh5tQrAYjQ498fiAGPAoOREFDDwPE7AFl47zd5P2ed
4ra+ruboAqMK8aXvhDUpfsi1AFEw9kD23PSSUVVLwBF0sORRy6ybch+3jqv+ya56
8D9+yTZ6K41bAQtg7zk3dumYhaB7S4URe68FbIwPprA939gz96PFnIk1JpdKVnFc
P4qEI44Cl1EwNsJUdb/LNShFa5HkECc8+LEXjrv6nDBm20OhAoHBAMLE9UANgoTI
ej84rYSIR5ZbY90pWnvyA9gkCHHwTCd4y0QLOX1eO/s9w48xY/eik9khGDQ+Nthw
DV0KIsLKsT6KgdCVocX8/vzDGa2kD1AKqqQoSAz1L6MipWqOK60T6NM6/14W7Odq
cPIbGTgkmPRUicbSIFMOrgh0DpNQozD9K1gL7GCRtBPRwbeIH59QEpYkjDd8KweI
0VaEtcAwu+ogUn119LCqgO2SCjOw7pA06/izi1t6CtNo3vxYJN4IOg==
-----END RSA PRIVATE KEY-----


Encrypted:
  b'b22e8f4d6fc0dd1ba6f1ffe4df099eac8d4c8d660a07f20cd50ade2bb02283427d23288388c8a145b130c0cd6bfebde5bba8a66c6dd8b834e062fc4b1e53e6cd70b472cea3
b7c1102888f02e0c2c52eb35a1a7b77df29ecea8d1fd9628a66c084996db355cc23d795559d71e7c0d08658d2fce67965d9950faff33898ea0d2630789d793f79bd6f606296e98
d2c92adf0bc20415a93926a786abbaa517cda801f1a937eb483778662cc999c36acaad3867069839c24e947896b620b63798cf47c2a345fc8d2884e8a2982b06ca14ef2d098f5e
0236065687168c0a6155ad941622aaa43a09e0068ba446702f5b805eb48d1d334f8d34604a1c1d8c964cb90efd80ed6cd2e75b261de918d951bbc900ca5815d1fdaf66551afaf4
d55d60390a25e5ecd15a37600a26af76658abf6766255d88f22ec458350082e29c5626ddee6882536e573c6fa23ee6ae1437c1d76be65480ea4b136f7fbdf67a0cd77ce93f7df2
c48a06252dc9db58637df63c076fffa1028fd92ace716a94a1d6'

Decrypted:
  b'Optical probing from the backside of an integrated circuit (IC) is a powerful failure analysis technique but raises serious security
concerns when in the hands of attackers. For instance, attacks using laser voltage probing (LVP) allow direct reading of sensitive information
being stored and/or processed in the IC.'


Execution TIme:   1.7898550033569336
```

Fig. 19: RSA code output (Execution time 1.78985 sec)

AES_CTR Code:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Nov  9 23:23:19 2020

@author: mdtamjidhossain
"""

import json
from base64 import b64encode, b64decode
from Cryptodome.Cipher import AES
from Cryptodome.Random import get_random_bytes
import hashlib

import time
#%%

start = time.time()

#%%

# Text color function
def colored(r, g, b, text):
    return "\033[38;2;{};{};{}m{} \033[38;2;255;255;255m".format(r, g, b, text)

#%%
# importing message from a file

plainTextLoc= '/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/plainText_long.txt'
# plainTextLoc= '/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/plainTextWithError.txt'
with open(plainTextLoc, 'r') as file:
    data = file.read().replace('\n', '')

message = data

plaintext = message
print(colored(255, 0, 0, 'Original Message (string):\n'), data, '\n')

#%%
# generating 32-bytes AES key

password ='cs654pass2020'

key = hashlib.sha256(password.encode()).digest()
print(colored(255, 0, 0, 'AES Key (32 bytes or 256 bits):\n'), key, '\n')
#%%
# Encryption using CTR mode

cipher = AES.new(key, AES.MODE_CTR)
ct_bytes = cipher.encrypt(plaintext.encode())
nonce = b64encode(cipher.nonce).decode('utf-8')
ct = b64encode(ct_bytes).decode('utf-8')
result = json.dumps({'nonce':nonce, 'ciphertext':ct})
print(colored(255,0,0,'Encrypted text:\n'),ct, '\n')
#%%
# Decryption using CTR mode

try:
    b64 = json.loads(result)
    nonce = b64decode(b64['nonce'])
    ct = b64decode(b64['ciphertext'])
    cipher = AES.new(key, AES.MODE_CTR, nonce=nonce)
    pt = cipher.decrypt(ct)
    print(colored(255,0,0,"Decrypted text:\n"), pt.decode(), '\n\n')
```

```
except(ValueError, KeyError):
    print("Incorrect decryption")
#%%

end = time.time()
print(colored(255,0,0, 'Execution TIme: '), end - start)
#%%
```

Output:

```
In [734]: runfile('/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/AES_CTR.py', wdir='/Users/mdtamjidhossain/Fall-2020/Courses/
CS654/Project')
Original Message (string):
  Optical probing from the backside of an integrated circuit (IC) is a powerful failure analysis technique but raises serious security
concerns when in the hands of attackers. For instance, attacks using laser voltage probing (LVP) allow direct reading of sensitive
information being stored and/or processed in the IC.

AES Key (32 bytes or 256 bits):
  b'3\xa3\xa0\xd8\x87\xbb\x9e\x11\x8f\xe6\x95\xf2\xb6;\xffo}\xfdEm\xb9\xabe\xbb\x08\xb1R$ha\xdel'

Encrypted text:
  ZvfOUYC9XwCbesKuwZfSMuXq5M7WVEr59xSTpVfa0nD0TNHgcogcpJS5WZOVZPnbosSkEgS7Omj7KYa2IjVxv4qa6/mJ
+CUT7ER2FlrqvYg4y9b00HjL9m9PGs33JQ2Rp2zoxHpAhPKogbs33+sTJV4EIa8t/mlw06DzaVm29g7bJw+mXT72Pel2494MSRjSrrjnbGsiqbeNcf
+28TijyQupk5ia8xxbxj3HgYAmXa84mQytyzVBsM5GEpxtC2weKXzK4PqeWz/sWQ66H3FLnFJsWjrqjmZGnJZMYobvgXn8qO1t+HrhYG49AOE+INyNhZPBjTx5Z/xDfU+UCOCTug
+KyewAF/0K3U1p8KkrSIhSK5o0lTyyGXWOIUHDDFr/sXDqFElwhALpcSmYctFPyT57dww2HYNurvPnfw==

Decrypted text:
  Optical probing from the backside of an integrated circuit (IC) is a powerful failure analysis technique but raises serious security
concerns when in the hands of attackers. For instance, attacks using laser voltage probing (LVP) allow direct reading of sensitive
information being stored and/or processed in the IC.


Execution TIme:   0.0015332698822021484
```

Fig. 20: AES code output with CTR mode (Execution time 0.001533 sec)

After analyzing both outputs from RSA and AES encryption technique below observations can be pointed out –

- The key length of RSA can be 1024/2048/3072 bits
- the key length of AES can be 128/192/256 bits
- In this project (for Part-2), AES mode CTR has been used
- Both schemes successfully encrypted and decrypted the plaintext
- **RSA consumed more time than AES. RSA-3072 takes around 1.78985 sec whereas AES-256 takes around 0.001533 sec. RSA spends most of the time in generating its keys. Key size of RSA is a major reason behind this.**

```
keyPair = RSA.generate(3072)
```