**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**](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
6. @author: mdtamjidhossain
7. """

10. #%%
11. # importing modules from library
13. **from** Cryptodome.Cipher **import** AES
15. **import** hashlib
16. **import** Padding
17. **import** binascii
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)
24. #%%
25. # encryption and decryption functions
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))
33. **return**(cipherList\_byte)
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))
40. **return**(plainTextList\_byte)
42. **def** encrypt2(blockList\_byte, key, mode, iv):
43. cipherList\_byte = []
44. encobj = AES.new(key,mode, iv)
45. **for** block **in** blockList\_byte:
46. cipherList\_byte.append(encobj.encrypt(block))
48. **return**(cipherList\_byte)
50. **def** decrypt2(cipherList\_byte,key, mode, iv):
51. plainTextList\_byte = []
52. encobj = AES.new(key,mode,iv)
53. **for** block **in** cipherList\_byte:
54. plainTextList\_byte.append(encobj.decrypt(block))
55. **return**(plainTextList\_byte)
57. **def** encrypt3(blockList\_byte, key, mode):
58. cipherList\_byte = []
59. encobj = AES.new(key,AES.MODE\_CTR)
60. nonce = encobj.nonce
61. **for** block **in** blockList\_byte:
62. cipherList\_byte.append(encobj.encrypt(block))
64. **return**(cipherList\_byte, nonce)
66. **def** decrypt3(cipherList\_byte,key, mode, nonce):
67. plainTextList\_byte = []
68. encobj = AES.new(key,mode, nonce =  nonce)
69. **for** block **in** cipherList\_byte:
70. plainTextList\_byte.append(encobj.decrypt(block))
71. **return**(plainTextList\_byte)
73. #%%
75. **def** plainTextToByte(plaintext):
76. plaintext = Padding.appendPadding(plaintext,blocksize=Padding.AES\_blocksize,mode=0)
77. plaintext\_byte = plaintext.encode('utf-8')
78. blockList\_byte = [plaintext\_byte[i:i+16] **for** i **in** range(0, len(plaintext\_byte), 16)]
79. **return** blockList\_byte
81. **def** byteToHexBlock(blockList\_byte):
82. blockList\_hex = []
83. **for** byte **in** blockList\_byte:
84. blockList\_hex.append(binascii.hexlify(byte).decode())
85. **return** blockList\_hex
87. #%%
89. # importing message from a file
90. plainTextLoc= '/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/plainText.txt'
91. # plainTextLoc= '/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/plainTextWithError.txt'
92. with open(plainTextLoc, 'r') as file:
93. data = file.read().replace('\n', '')
95. # plaintext = data.encode('utf-8')
96. message = data
98. plaintext = message
99. #%%
101. #1. With AES as the encryption / decryption algorithm, generate your own key.
103. password ='cs654pass2020'
104. ival=12
106. key = hashlib.sha256(password.encode()).digest()
108. iv= hex(ival)[2:8].zfill(16)

111. **print**(colored(255, 0, 0, 'AES Key (32 bytes or 256 bits):'))
112. **print**(str(key) + '\n')
114. **print**(colored(255, 0, 0, 'AES initialization vector (IV) :'))
115. **print**(str(iv) + '\n')
117. #%%
118. blockList\_byte = []
119. blockList\_hex = []
121. #%%
122. # encrypt and decrypt using ECB, CBC, CFB, OFB, CTR
123. **print**(colored(255, 0, 0, 'Original Message (string):'))
124. **print**(str(data) + '\n')
126. blockList\_byte = plainTextToByte(data)
127. **print**(colored(255, 0, 0, 'Original Message after padding (bytes):'))
128. **print**(str(blockList\_byte) + '\n')
130. blockList\_hex = byteToHexBlock(blockList\_byte)
131. **print**(colored(255, 0, 0, 'Original Message after padding (hex):'))
132. **print**(str(blockList\_hex) + '\n')
133. **print**(colored(255, 0, 0, 'Encrypt and Decrypt using ECB, CBC, CFB, OFB, CTR :'))
135. #--------------------ECB starts---------------------------------------------
136. cipherList\_byte = encrypt(blockList\_byte,key, AES.MODE\_ECB)
137. blockList\_hex = byteToHexBlock(cipherList\_byte)
138. **print** (colored(0, 255, 0, "  CipherBlock (ECB):\t")+str(blockList\_hex))
139. **print** (colored(0, 255, 0, "  Ciphertext (ECB):\t")+ str(''.join(blockList\_hex)))

142. plainTextList\_byte = decrypt(cipherList\_byte,key,AES.MODE\_ECB)
143. plainTextList\_str = []
144. **for** plaintext **in** plainTextList\_byte:
145. **try**:
146. plainTextList\_str.append(Padding.removePadding(plaintext.decode(),mode=0))
147. **except**:
148. plainTextList\_str.append(plaintext.decode())
149. **print** (colored(0, 255, 0, "  PlaintextBlock (ECB):\t")+str(plainTextList\_str))
150. **print** (colored(0, 255, 0, "  Plaintext (ECB):\t")+str(''.join(plainTextList\_str))+ '\n')
151. #--------------------ECB ends---------------------------------------------


155. #--------------------CBC starts---------------------------------------------
156. cipherList\_byte = encrypt2(blockList\_byte,key, AES.MODE\_CBC, iv.encode())
157. blockList\_hex = byteToHexBlock(cipherList\_byte)
158. **print** (colored(235, 204, 52, "  CipherBlock (CBC):\t   ")+str(blockList\_hex))
159. **print** (colored(235, 204, 52, "  Ciphertext (CBC):\t   ")+ str(''.join(blockList\_hex)))

162. plainTextList\_byte = decrypt2(cipherList\_byte,key,AES.MODE\_CBC,iv.encode())
163. plainTextList\_str = []
164. **for** plaintext **in** plainTextList\_byte:
165. **try**:
166. plainTextList\_str.append(Padding.removePadding(plaintext.decode(),mode=0))
167. **except**:
168. plainTextList\_str.append(plaintext.decode())
169. **print** (colored(235, 204, 52, "  PlaintextBlock (CBC):  ")+str(plainTextList\_str))
170. **print** (colored(235, 204, 52, "  Plaintext (CBC):\t   ")+str(''.join(plainTextList\_str))+ '\n')
171. #--------------------CBC ends---------------------------------------------
173. #--------------------CFB starts---------------------------------------------
174. cipherList\_byte = encrypt2(blockList\_byte,key, AES.MODE\_CFB, iv.encode())
175. blockList\_hex = byteToHexBlock(cipherList\_byte)
176. **print** (colored(52, 201, 235, "  CipherBlock (CFB):\t   ")+str(blockList\_hex))
177. **print** (colored(52, 201, 235, "  Ciphertext (CFB):\t   ")+ str(''.join(blockList\_hex)))

180. plainTextList\_byte = decrypt2(cipherList\_byte,key,AES.MODE\_CFB,iv.encode())
181. plainTextList\_str = []
182. **for** plaintext **in** plainTextList\_byte:
183. **try**:
184. plainTextList\_str.append(Padding.removePadding(plaintext.decode(),mode=0))
185. **except**:
186. plainTextList\_str.append(plaintext.decode())
187. **print** (colored(52, 201, 235, "  PlaintextBlock (CFB):  ")+str(plainTextList\_str))
188. **print** (colored(52, 201, 235, "  Plaintext (CFB):\t   ")+str(''.join(plainTextList\_str)) + '\n')
189. #--------------------CFB ends---------------------------------------------
191. #--------------------OFB starts---------------------------------------------
192. cipherList\_byte = encrypt2(blockList\_byte,key, AES.MODE\_OFB, iv.encode())
193. blockList\_hex = byteToHexBlock(cipherList\_byte)
194. **print** (colored(183, 52, 235, "  CipherBlock (OFB):\t   ")+str(blockList\_hex))
195. **print** (colored(183, 52, 235, "  Ciphertext (OFB):\t   ")+ str(''.join(blockList\_hex)))

198. plainTextList\_byte = decrypt2(cipherList\_byte,key,AES.MODE\_OFB,iv.encode())
199. plainTextList\_str = []
200. **for** plaintext **in** plainTextList\_byte:
201. **try**:
202. plainTextList\_str.append(Padding.removePadding(plaintext.decode(),mode=0))
203. **except**:
204. plainTextList\_str.append(plaintext.decode())
205. **print** (colored(183, 52, 235, "  PlaintextBlock (OFB):  ")+str(plainTextList\_str))
206. **print** (colored(183, 52, 235, "  Plaintext (OFB):\t   ")+str(''.join(plainTextList\_str))+ '\n')
207. #--------------------OFB ends---------------------------------------------

210. #--------------------CTR starts---------------------------------------------
211. cipherList\_byte, nonce = encrypt3(blockList\_byte,key, AES.MODE\_CTR)
212. blockList\_hex = byteToHexBlock(cipherList\_byte)
213. **print** (colored(235, 125, 52, "  CipherBlock (CTR):\t   ")+str(blockList\_hex))
214. **print** (colored(235, 125, 52, "  Ciphertext (CTR):\t   ")+ str(''.join(blockList\_hex)))

217. plainTextList\_byte = decrypt3(cipherList\_byte,key,AES.MODE\_CTR,nonce)
218. plainTextList\_str = []
219. **for** plaintext **in** plainTextList\_byte:
220. **try**:
221. plainTextList\_str.append(Padding.removePadding(plaintext.decode(),mode=0))
222. **except**:
223. plainTextList\_str.append(plaintext.decode())
224. **print** (colored(235, 125, 52, "  PlaintextBlock (CTR):  ")+str(plainTextList\_str))
225. **print** (colored(235, 125, 52, "  Plaintext (CTR):\t   ")+str(''.join(plainTextList\_str))+ '\n')
226. nonce = ''
227. #--------------------CTR ends---------------------------------------------

230. #%%

**Output**

Text

Description automatically generatedFig. 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).

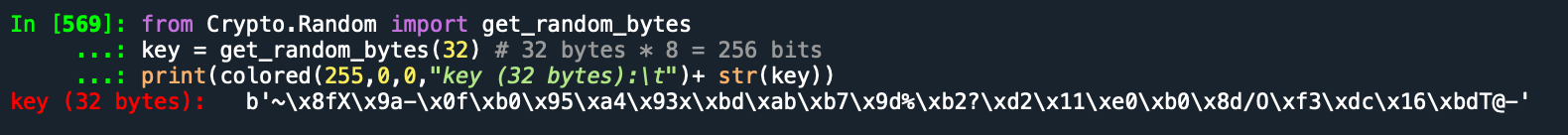


Fig. 2: Random AES key generation

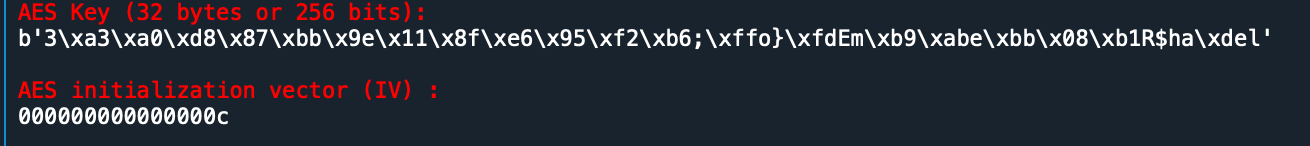
We can also generate 32-bytes key using SHA-256 or PBKDF2 algorithm. For simplicity, in this project, SHA-256 has been used to generate a 32-bytes (256-bits) key using a password (see Fig. 3). This password can be –

* 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.

 **Text

Description automatically generated**

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.

Text

Description automatically generated

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.
* Graphical user interface, text

  Description automatically generated*‘encrypt3’* and *‘decrypt3’* functions are used for CTR mode where nonce is required.

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.

*Text

Description automatically generated*

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).

Text

Description automatically generatedText

Description automatically generated

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:**

**Text

Description automatically generated Code:**

Fig. 9: Code of ECB Mode

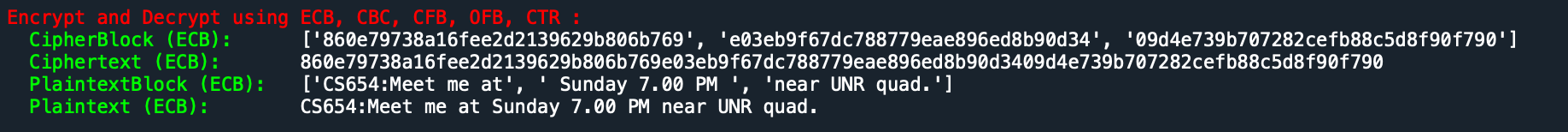
** Output:**

Fig. 10: Output of ECB Mode

**CBC Mode:**

**A screenshot of a cell phone

Description automatically generated Code:**

Fig. 11: Code of CBC Mode

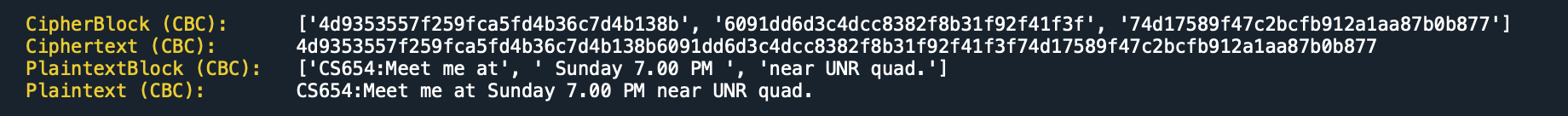
** Output:**

Fig. 12: Output of CBC Mode

**CFB Mode:**

**A screenshot of text

Description automatically generated Code:**

Fig. 13: Code of CFB Mode

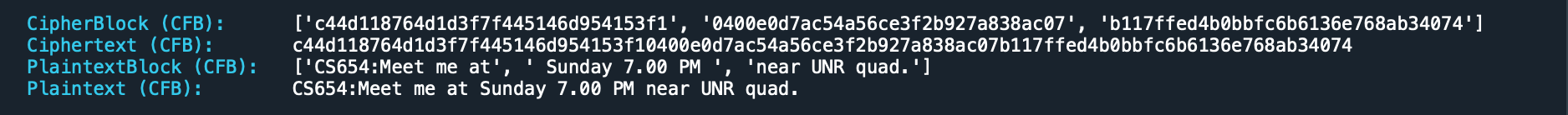
** Output:**

Fig. 14: Output of CFB Mode

**OFB Mode:**

**A screenshot of text

Description automatically generated Code:**

Fig. 15: Code of OFB Mode

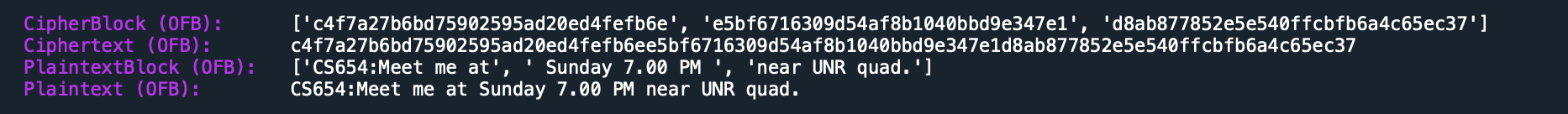
** Output:**

Fig. 16: Output of OFB Mode

**CTR Mode:**

**Text

Description automatically generated Code:**

Fig. 15: Code of CTR Mode

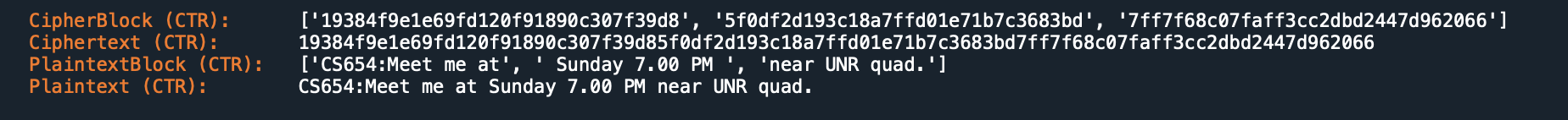
** Output:**

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.’

**Text

Description automatically generatedOutput:**

Figure 17: Output with error message

Text

Description automatically generatedWe 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.

Text

Description automatically generatedOriginal message

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**

1. #!/usr/bin/env python3
2. # -\*- coding: utf-8 -\*-
3. """
4. Created on Wed Nov 11 12:04:42 2020
6. @author: mdtamjidhossain
7. """
8. #%%
9. **from** Cryptodome.PublicKey **import** RSA
10. **from** Cryptodome.Cipher **import** PKCS1\_OAEP
11. **import** binascii
13. **import** time
14. #%%
16. start = time.time()
17. #%%
19. # Text color function
20. **def** colored(r, g, b, text):
21. **return** "\033[38;2;{};{};{}m{} \033[38;2;255;255;255m".format(r, g, b, text)
23. #%%
24. # importing message from a file
26. plainTextLoc= '/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/plainText\_long.txt'
27. # plainTextLoc= '/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/plainTextWithError.txt'
28. with open(plainTextLoc, 'r') as file:
29. data = file.read().replace('\n', '')
31. message = data
33. plaintext = message
34. #%%
36. keyPair = RSA.generate(3072)
38. pubKey = keyPair.publickey()
39. **print**(colored(255, 0, 0, "Public key:"))
40. **print**(f"(n={hex(pubKey.n)}, e={hex(pubKey.e)})" + '\n')
42. pubKeyPEM = pubKey.exportKey()
43. **print**(pubKeyPEM.decode('ascii')+ '\n')
45. **print**(colored(255, 0, 0, "Private key:"))
46. **print**(f"(n={hex(pubKey.n)}, d={hex(keyPair.d)})"+'\n')
47. privKeyPEM = keyPair.exportKey()
48. **print**(privKeyPEM.decode('ascii')+'\n\n')
49. #%%
50. # Encryption
52. encryptor = PKCS1\_OAEP.new(pubKey)
53. encrypted = encryptor.encrypt(plaintext.encode())
54. **print**(colored(255,0,0, "Encrypted:\n"), binascii.hexlify(encrypted), '\n')
55. #%%
56. # Decryption
58. decryptor = PKCS1\_OAEP.new(keyPair)
59. decrypted = decryptor.decrypt(encrypted)
60. **print**(colored(255,0,0,'Decrypted:\n'), decrypted, '\n\n')
61. #%%
63. end = time.time()
64. **print**(colored(255,0,0, 'Execution TIme: '), end - start)
65. #%%

**Output**

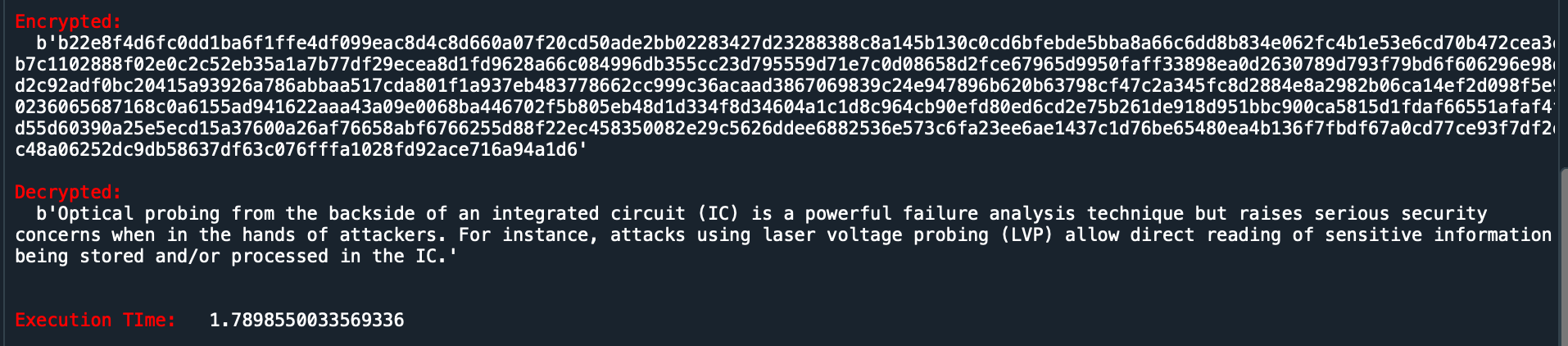
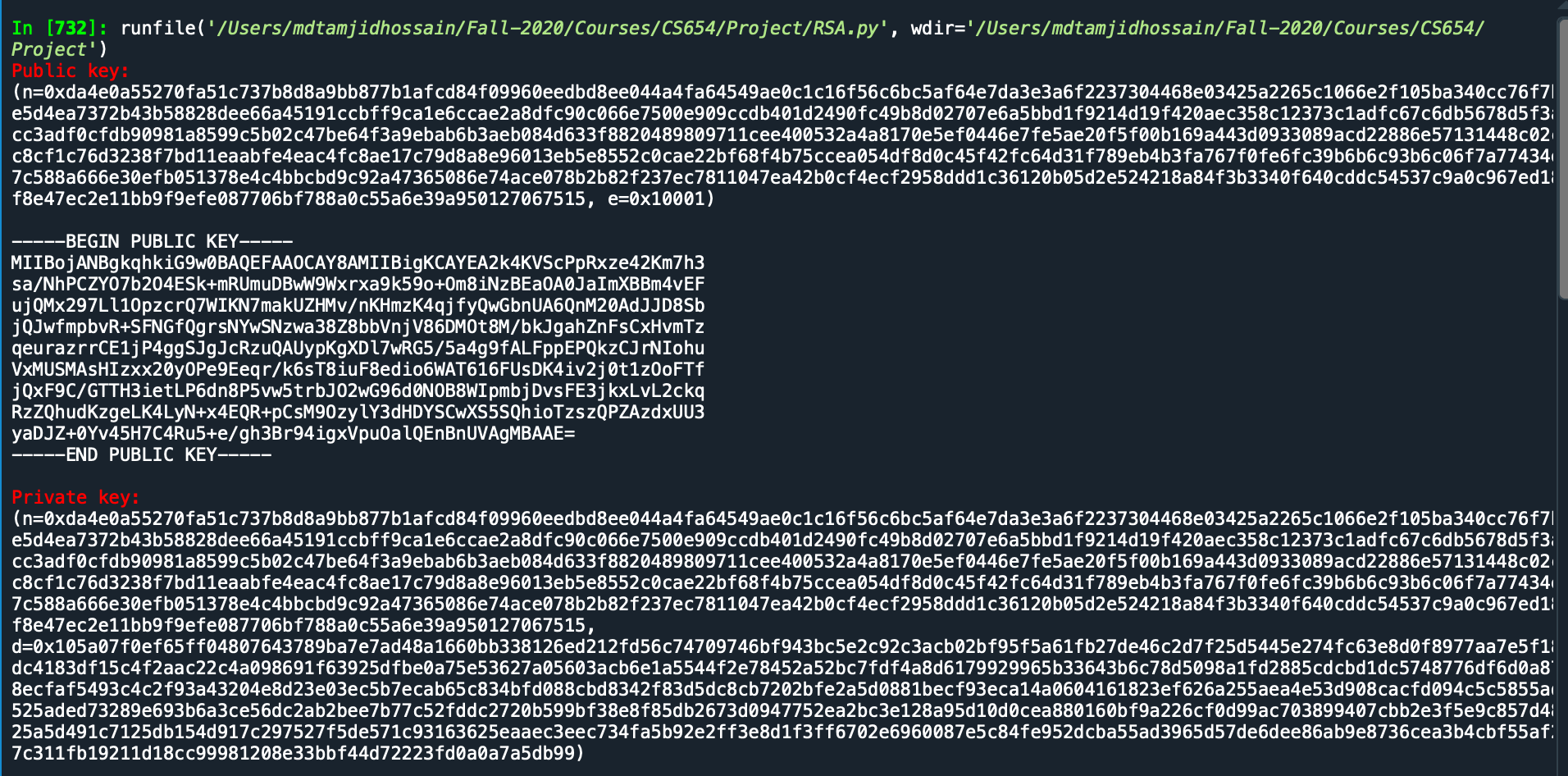


Fig. 19: RSA code output (Execution time 1.78985 sec)

**AES\_CTR Code:**

1. #!/usr/bin/env python3
2. # -\*- coding: utf-8 -\*-
3. """
4. Created on Mon Nov  9 23:23:19 2020
6. @author: mdtamjidhossain
7. """
9. **import** json
10. **from** base64 **import** b64encode, b64decode
11. **from** Cryptodome.Cipher **import** AES
12. **from** Cryptodome.Random **import** get\_random\_bytes
13. **import** hashlib
15. **import** time
16. #%%
18. start = time.time()
20. #%%
22. # Text color function
23. **def** colored(r, g, b, text):
24. **return** "\033[38;2;{};{};{}m{} \033[38;2;255;255;255m".format(r, g, b, text)
26. #%%
27. # importing message from a file
29. plainTextLoc= '/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/plainText\_long.txt'
30. # plainTextLoc= '/Users/mdtamjidhossain/Fall-2020/Courses/CS654/Project/plainTextWithError.txt'
31. with open(plainTextLoc, 'r') as file:
32. data = file.read().replace('\n', '')
34. message = data
36. plaintext = message
37. **print**(colored(255, 0, 0, 'Original Message (string):\n'), data, '\n')
39. #%%
40. # generating 32-bytes AES key
42. password ='cs654pass2020'
44. key = hashlib.sha256(password.encode()).digest()
45. **print**(colored(255, 0, 0, 'AES Key (32 bytes or 256 bits):\n'), key, '\n')
46. #%%
47. # Encryption using CTR mode
49. cipher = AES.new(key, AES.MODE\_CTR)
50. ct\_bytes = cipher.encrypt(plaintext.encode())
51. nonce = b64encode(cipher.nonce).decode('utf-8')
52. ct = b64encode(ct\_bytes).decode('utf-8')
53. result = json.dumps({'nonce':nonce, 'ciphertext':ct})
54. **print**(colored(255,0,0,'Encrypted text:\n'),ct, '\n')
55. #%%
56. # Decryption using CTR mode
58. **try**:
59. b64 = json.loads(result)
60. nonce = b64decode(b64['nonce'])
61. ct = b64decode(b64['ciphertext'])
62. cipher = AES.new(key, AES.MODE\_CTR, nonce=nonce)
63. pt = cipher.decrypt(ct)
64. **print**(colored(255,0,0,"Decrypted text:\n"), pt.decode(), '\n\n')
65. **except**(ValueError, KeyError):
66. **print**("Incorrect decryption")
67. #%%
69. end = time.time()
70. **print**(colored(255,0,0, 'Execution TIme: '), end - start)
71. #%%

**Output:**

**Text

Description automatically generated**

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)**