CSE 350 Network Security
Programming Assignment Number 2

Akshat Saini 2020019 Nakul Thureja 2020528

Project Number = (0019 + 0528) mod 2 = 1

Project 1: You are required to develop a program to encrypt (and similarly decrypt) a 128-bit plaintext using AES that uses keys of size 128 bit, and 10 rounds (repeat, 10 rounds). Instead of using an available library, I insist that you program each and every element of each of the 10 rounds of AES (and that means Substitute bytes, shift-rows, etc., etc., and generation of sub-keys,
etc.). Having done that, with at least TWO pairs of <128-bit plaintext, ciphertext>:

a. Verify that the ciphertext when decrypted will yield the original plaintext
b. Verify that the output of 1st encryption round is same as output of the 9th decryption round as illustrated below
c. Verify that the output of 9th encryption round is same as output of the 1st decryption round as illustrated below.

Input:

- A 128 bit plaintext which is in hexadecimal format.
- A 128 bit key which is in hexadecimal format.

For the purpose of testing I have generated random Inputs

Output:

- A 128 bit Ciphertext which is in hexadecimal format.
- A 128 bit plaintext which is in hexadecimal format (decyphered from ciphertext generated).
- Round 1 and 9 encryption output for encryption and decryption

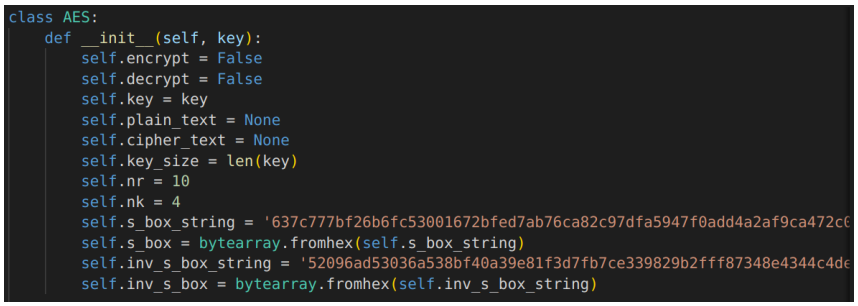
Command to run:
python3 A2_2020019_2020528.py

System Designed

The system designed has a class AES 2 major functions, namely **encrypter and decrypter**, which perform 128 bit AES **encryption** with a **fixed key length of 128 bit.** Along with this there are some helper functions regarding that lets look into it one-by-one.

1. **Class AES**

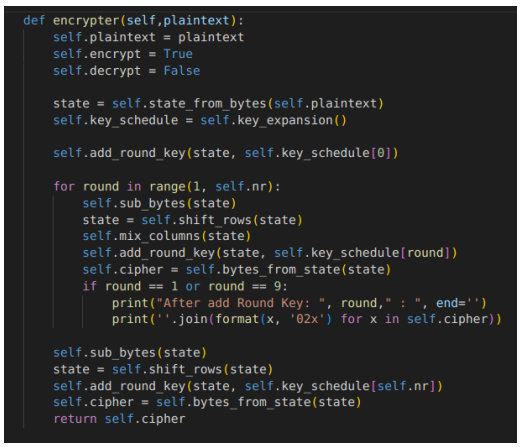
   Initially class is initialized with the 128 bit key and some variables are set which contains i.e. sbox and the inverse_sbox (same as the one given in class).



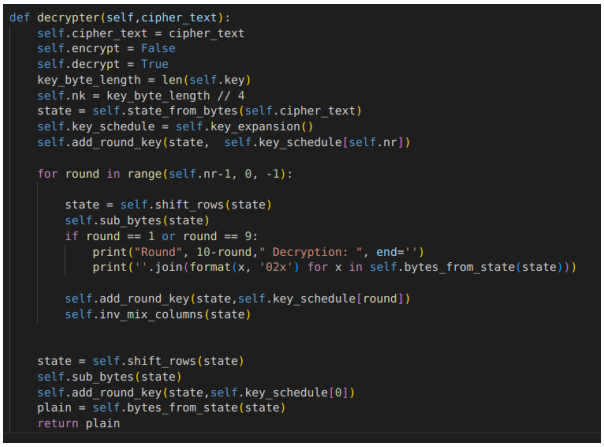
```
class AES:
    def __init__(self, key):
        self.encrypt = False
        self.decrypt = False
        self.key = key
        self.plain_text = None
        self.cipher_text = None
        self.key_size = len(key)
        self.nr = 10
        self.nk = 4
        self.s_box_string = '637c777bf26b6fc53001672bfed7ab76ca82c97dfa5947f0add4a2af9ca472c0
        self.s_box = bytearray.fromhex(self.s_box_string)
        self.inv_s_box_string = '52096ad53036a538bf40a39e81f3d7fb7ce339829b2fff87348e4344c4de
        self.inv_s_box = bytearray.fromhex(self.inv_s_box_string)
```

2. **Encrypter**

   The encrypt function is responsible for using the key to encrypt the raw messages using AES algorithm.



```
def encrypter(self,plaintext):
    self.plaintext = plaintext
    self.encrypt = True
    self.decrypt = False

    state = self.state_from_bytes(self.plaintext)
    self.key_schedule = self.key_expansion()

    self.add_round_key(state, self.key_schedule[0])

    for round in range(1, self.nr):
        self.sub_bytes(state)
        state = self.shift_rows(state)
        self.mix_columns(state)
        self.add_round_key(state, self.key_schedule[round])
        self.cipher = self.bytes_from_state(state)
        if round == 1 or round == 9:
            print("After add Round Key: ", round," : ", end='')
            print(''.join(format(x, '02x') for x in self.cipher))

    self.sub_bytes(state)
    state = self.shift_rows(state)
    self.add_round_key(state, self.key_schedule[self.nr])
    self.cipher = self.bytes_from_state(state)
    return self.cipher
```

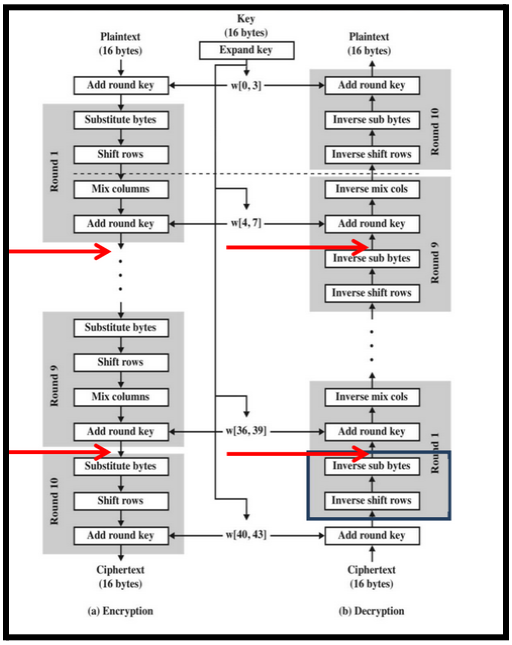


```
def decrypter(self,cipher_text):
    self.cipher_text = cipher_text
    self.encrypt = False
    self.decrypt = True
    key_byte_length = len(self.key)
    self.nk = key_byte_length // 4
    state = self.state_from_bytes(self.cipher_text)
    self.key_schedule = self.key_expansion()
    self.add_round_key(state,  self.key_schedule[self.nr])

    for round in range(self.nr-1, 0, -1):

        state = self.shift_rows(state)
        self.sub_bytes(state)
        if round == 1 or round == 9:
            print("Round", 10-round," Decryption: ", end='')
            print(''.join(format(x, '02x') for x in self.bytes_from_state(state)))

        self.add_round_key(state,self.key_schedule[round])
        self.inv_mix_columns(state)


    state = self.shift_rows(state)
    self.sub_bytes(state)
    self.add_round_key(state,self.key_schedule[0])
    plain = self.bytes_from_state(state)
    return plain
```

3. **Decrypt**

   The decrypt function is responsible for using the key to decrypt the raw messages using AES algorithm.

To encrypt and decrypt we will follow the following flow chart given below in class:



Helper Functions Defined:

- key_expansion: The function to generate 10 round keys for AES algorithm using the initial key. The function will return an array with 11 keys.

```
def key_expansion(self):
    w = self.state_from_bytes(self.key)

    for i in range(self.nk, self.nk * (self.nr + 1)):
        temp = w[i-1]
        if i % self.nk == 0:
            temp = temp[1:] + temp[:1]
            temp = self.xor_bytes(self.sub_word((temp)), self.rcon(i // self.nk))
        elif self.nk > 6 and i % self.nk == 4:
            temp = self.sub_word(temp)
        temp = self.xor_bytes(temp, w[i - self.nk])
        w.append(temp)

    return [w[i:i+4] for i in range(0,len(w),4)]
```

```
def add_round_key(self,state, round_key):
    for r in range(len(state)):
        l = []
        for j in range(len(state[0])):
            l.append(state[r][j] ^ round_key[r][j])
        state[r] = l
```

- add_round_key: Function to take XOR of the plaintext with round key.

- sub_bytes: function to replace the bytes from the s_box lookup table for encryption and replace the bytes from the inverse_s_box lookup table for decryption.

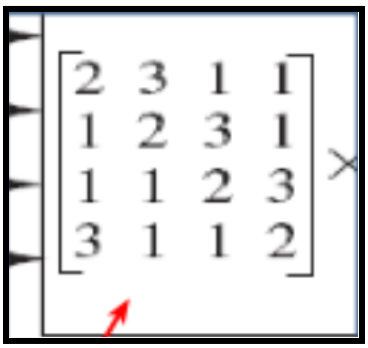
```
def sub_bytes(self,state):
    if self.encrypt:
        for r in range(len(state)):
            state[r] = [self.s_box[state[r][c]] for c in range(len(state[0]))]
    if self.decrypt:
        for r in range(len(state)):
            state[r] = [self.inv_s_box[state[r][c]] for c in range(len(state[0]))]
```

```
def shift_rows(self,state):
    state = self.transpose(state)
    if self.encrypt:
        for i in range(1, len(state)):
            state[i] = state[i][i:] + state[i][:i]
    if self.decrypt:
        for i in range(1, len(state)):
            state[i] = state[i][-i:] + state[i][:-i]
    state = self.transpose(state)
    return state
```

- Shift_rows: function to shift rows by i. For encryption it shifts left for decryption it shifts right. Note: since I have stored the matrix row-wise rather than column-wise, the matrix needs to be transposed before shifting.

- mix columns: function to multiply the matrix with MC matrix, but since I have stored the matrix row wise rather than column wise I am able to simplify the multiplication for each column.

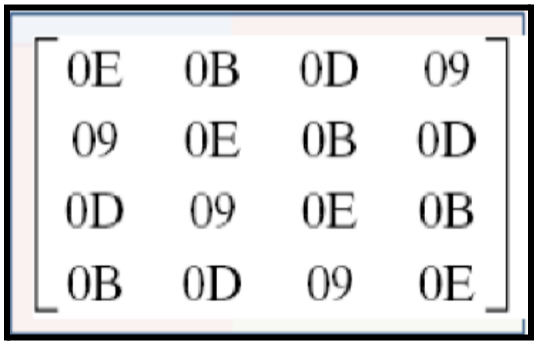
```
def mix_column(self,col):
    c_0 = col[0]
    all_xor = col[0]
    all_xor = all_xor ^ col[1]
    all_xor = all_xor ^ col[2]
    all_xor = all_xor ^ col[3]
    col[0] ^= all_xor ^ self.galois(col[1] ^ col[0])
    col[1] ^= all_xor ^ self.galois(col[2] ^ col[1])
    col[2] ^= all_xor ^ self.galois(col[3] ^ col[2])
    col[3] ^= all_xor ^ self.galois(col[3] ^ c_0)

def mix_columns(self,state):
    for r in state:
        self.mix_column(r)
```



MC Matrix used as given in lectures.

- inverse mix columns: function to multiply the matrix with inverse MC matrix, but since I have stored the matrix row wise rather than column wise I am able to simplify the multiplication for each column as given below.

```
def inv_mix_column(self,col):
    u_ = self.galois(col[2] ^ col[0])
    u = self.galois(u_)
    col[0] ^= u
    col[2] ^= u
    v_ = self.galois(col[3] ^ col[1])
    v = self.galois(v_)
    col[1] ^= v
    col[3] ^= v

def inv_mix_columns(self,state):
    for r in state:
        self.inv_mix_column(r)
    self.mix_columns(state)
```



Inverse MC Matrix used

Sample Test Cases:

```
if __name__ == "__main__":
    import random
    plaintext_hexes = ['0123456789abcdeffedcba9876543210','00112233445566778899aabbccddeeff']
    key_hexes = ['0f1571c947d9e8590cb7add6af7f6798','000102030405060708090a0b0c0d0e0f']
    ciphertext_hexes = ['ff0b844a0853bf7c6934ab4364148fb9','69c4e0d86a7b0430d8cdb78070b4c55a']

    for i in range(2):
        print("_____Test Case ",i+1,"_____")
        plaintext_hex = plaintext_hexes[i]
        plaintext = bytearray.fromhex(plaintext_hex)

        print("Plaintext : ", plaintext_hex)

        key_hex = key_hexes[i]
        key = bytearray.fromhex(key_hex)
        print("Key : ",key_hex)

        ciphertext = AES(key).encrypter(plaintext)
        ciphertext_hex = ''.join(format(x, '02x') for x in ciphertext)

        print("Ciphertext : ", ciphertext_hex)

        recovered_plaintext = AES(key).decrypter(ciphertext)
        print("Recovered Ciphertext : ", end='')
        print(''.join(format(x, '02x') for x in recovered_plaintext))
        assert (recovered_plaintext == plaintext)
        assert (ciphertext_hex == ciphertext_hexes[i])
        print()
```

- Key and Plaintext are 128 bit bytearray.
- We encrypt it using our AES class with the given key and get the ciphertext.
- We then decrypt it using our AES class AES class with the given key and get the recovered plaintext.
- Finally, we assert the recovered plaintext is same as the initial plaintext, and assert that expected_ciphertext is same as the ciphertext generated therefore the encryption and decryption are working fine.

Output:

```
_____Test Case  1 _____
Plaintext :  0123456789abcdeffedcba9876543210
Key :  0f1571c947d9e8590cb7add6af7f6798
After add Round Key:  1  : 657470750fc7ff3fc0e8e8ca4dd02a9c
After add Round Key:  9  : cca104a13e678500ff59025f3bafaa34
Ciphertext :  ff0b844a0853bf7c6934ab4364148fb9
Round 1  Decryption: cca104a13e678500ff59025f3bafaa34
Round 9  Decryption: 657470750fc7ff3fc0e8e8ca4dd02a9c
Recovered Ciphertext : 0123456789abcdeffedcba9876543210

_____Test Case  2 _____
Plaintext :  00112233445566778899aabbccddeeff
Key :  000102030405060708090a0b0c0d0e0f
After add Round Key:  1  : 89d810e8855ace682d1843d8cb128fe4
After add Round Key:  9  : bd6e7c3df2b5779e0b61216e8b10b689
Ciphertext :  69c4e0d86a7b0430d8cdb78070b4c55a
Round 1  Decryption: bd6e7c3df2b5779e0b61216e8b10b689
Round 9  Decryption: 89d810e8855ace682d1843d8cb128fe4
Recovered Ciphertext : 00112233445566778899aabbccddeeff
```