Cryptography and Network Security Lab

Assignment 7  
Implementation and Understanding of Advanced Encryption Standard (AES) Cipher

2019BTECS00058  
Devang K

Batch: B2

Title: Implementation and Understanding of Advanced Encryption Standard (AES)

Aim: To Study, Implement and Demonstrate the Advanced Encryption Standard (AES)

* Part A- Implementation of AES using Virtual Lab
* Part B- Implementation of AES using C/C++/Java/Python or any other programming language

Theory:

The Advanced Encryption Standard (AES), also known by its original name Rijndael, as a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001.

AES is a variant of the Rijndael block cipher developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, who submitted a proposal to NIST during the AES selection process. Rijndael is a family of ciphers with different key and block sizes. For AES, NIST selected three members of the Rijndael family, each with a block size of 128 bits, but three different key lengths: 128, 192 and 256 bits.

AES has been adopted by the U.S. government. It supersedes the Data Encryption Standard (DES),[7] which was published in 1977. The algorithm described by AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data.

Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by the U.S National Institute of Standards and Technology (NIST) in 2001. AES is widely used today as it is a much stronger than DES and triple DES despite being harder to implement.

Points to remember

* AES is a block cipher.
* The key size can be 128/192/256 bits.
* Encrypts data in blocks of 128 bits each.

That means it takes 128 bits as input and outputs 128 bits of encrypted cipher text as output. AES relies on substitution-permutation network principle which means it is performed using a series of linked operations which involves replacing and shuffling of the input data.

The number of rounds depends on the key length as follows :

* 128 bit key – 10 rounds
* 192 bit key – 12 rounds
* 256 bit key – 14 rounds

A Key Schedule algorithm is used to calculate all the round keys from the key. So, the initial key is used to create many different round keys which will be used in the corresponding round of the encryption.

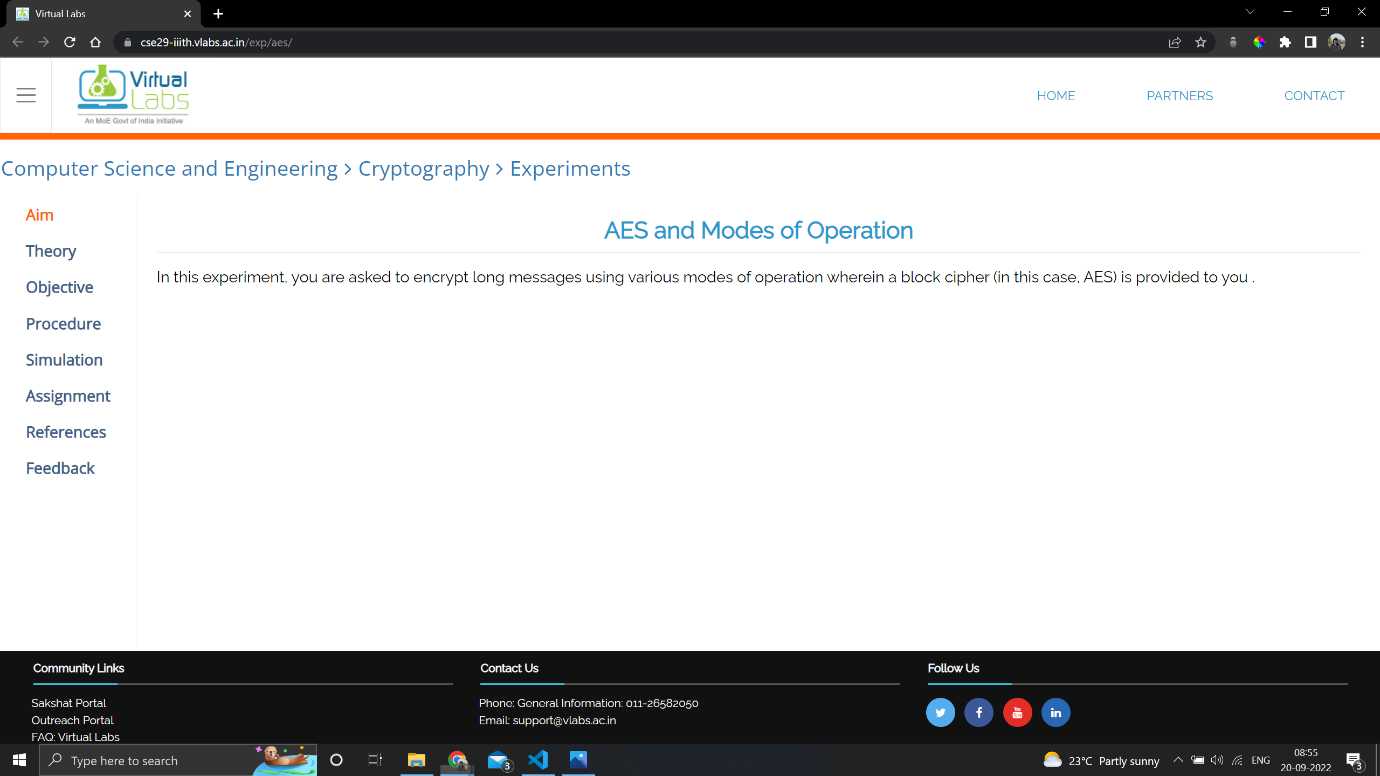
Each round comprises of 4 steps :

* SubBytes
* ShiftRows
* MixColumns
* Add Round Key

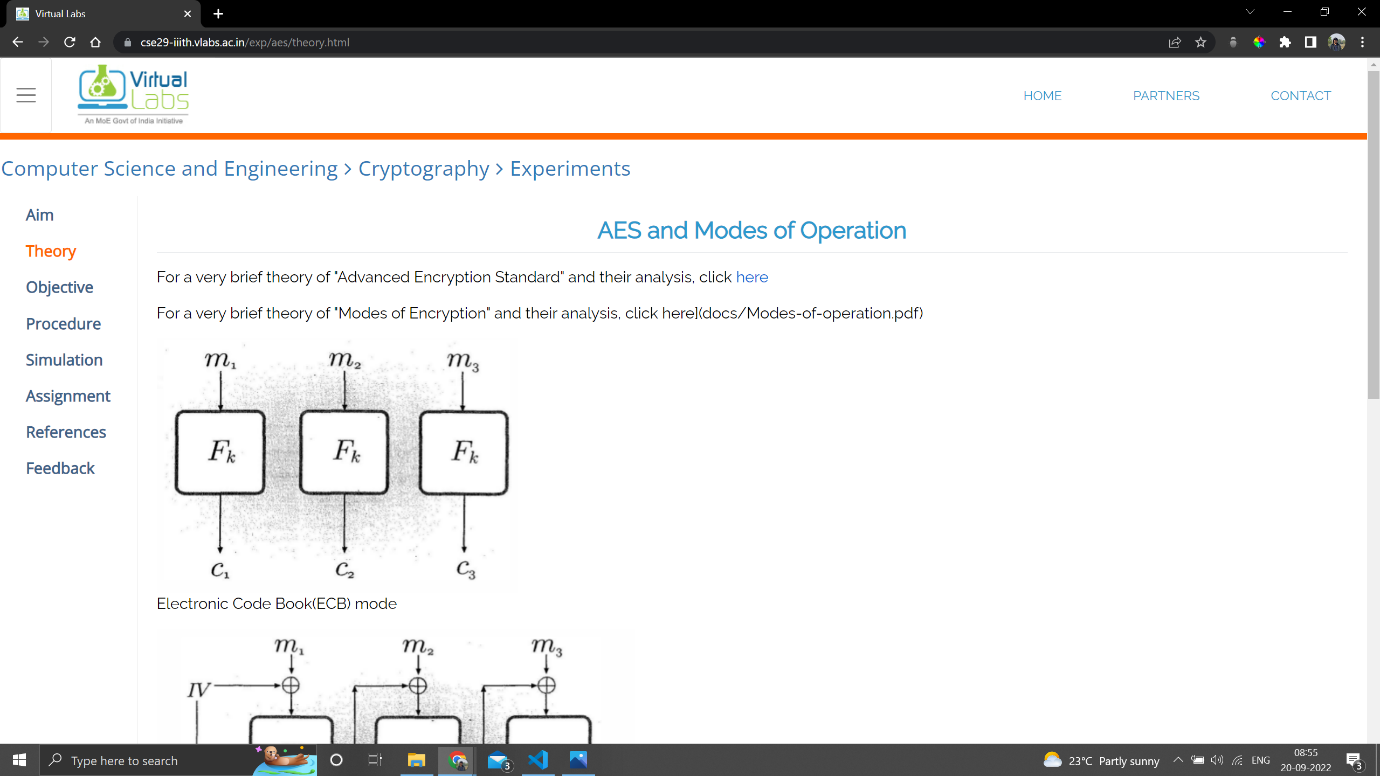
The last round doesn’t have the MixColumns round.

AES instruction set is now integrated into the CPU (offers throughput of several GB/s)to improve the speed and security of applications that use AES for encryption and decryption. Even though its been 20 years since its introduction we have failed to break the AES algorithm as it is infeasible even with the current technology. Till date the only vulnerability remains in the implementation of the algorithm.

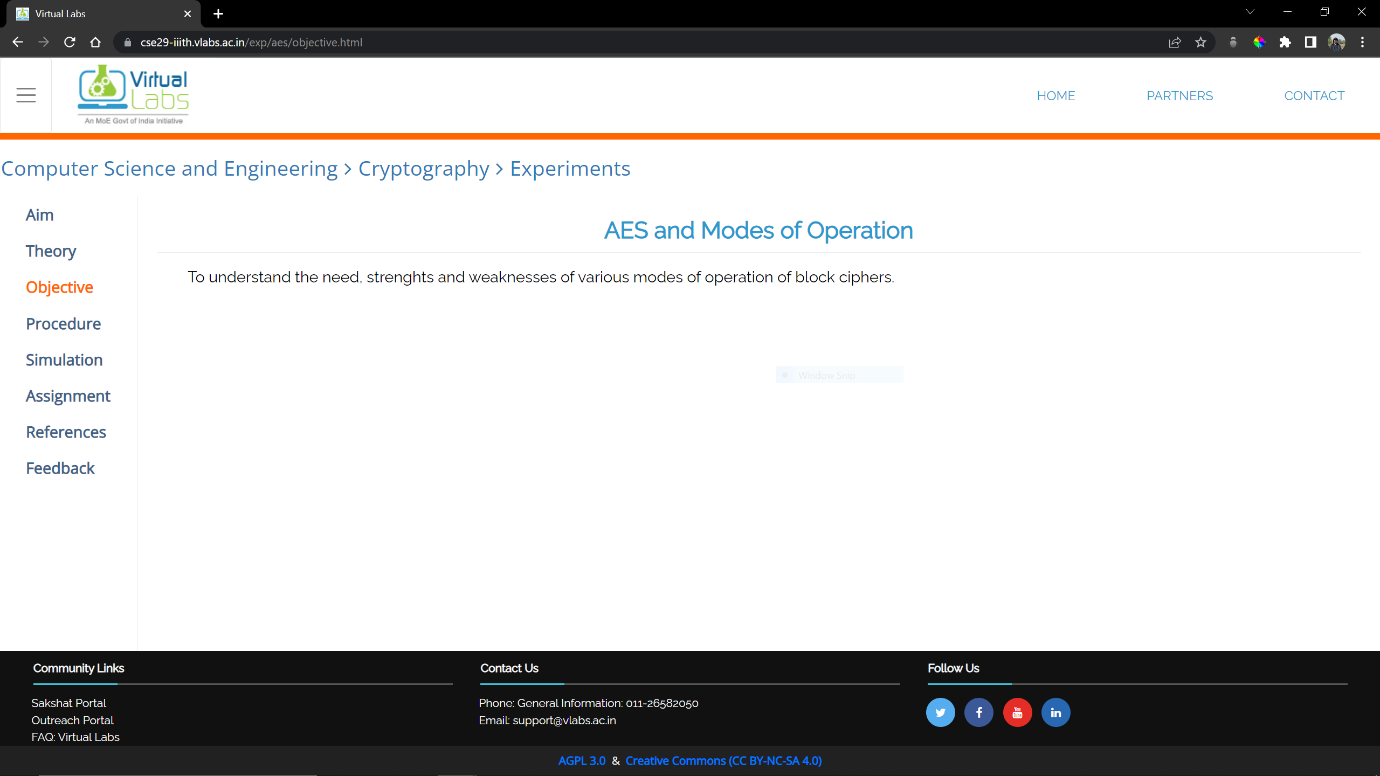
V-Lab Implementation:



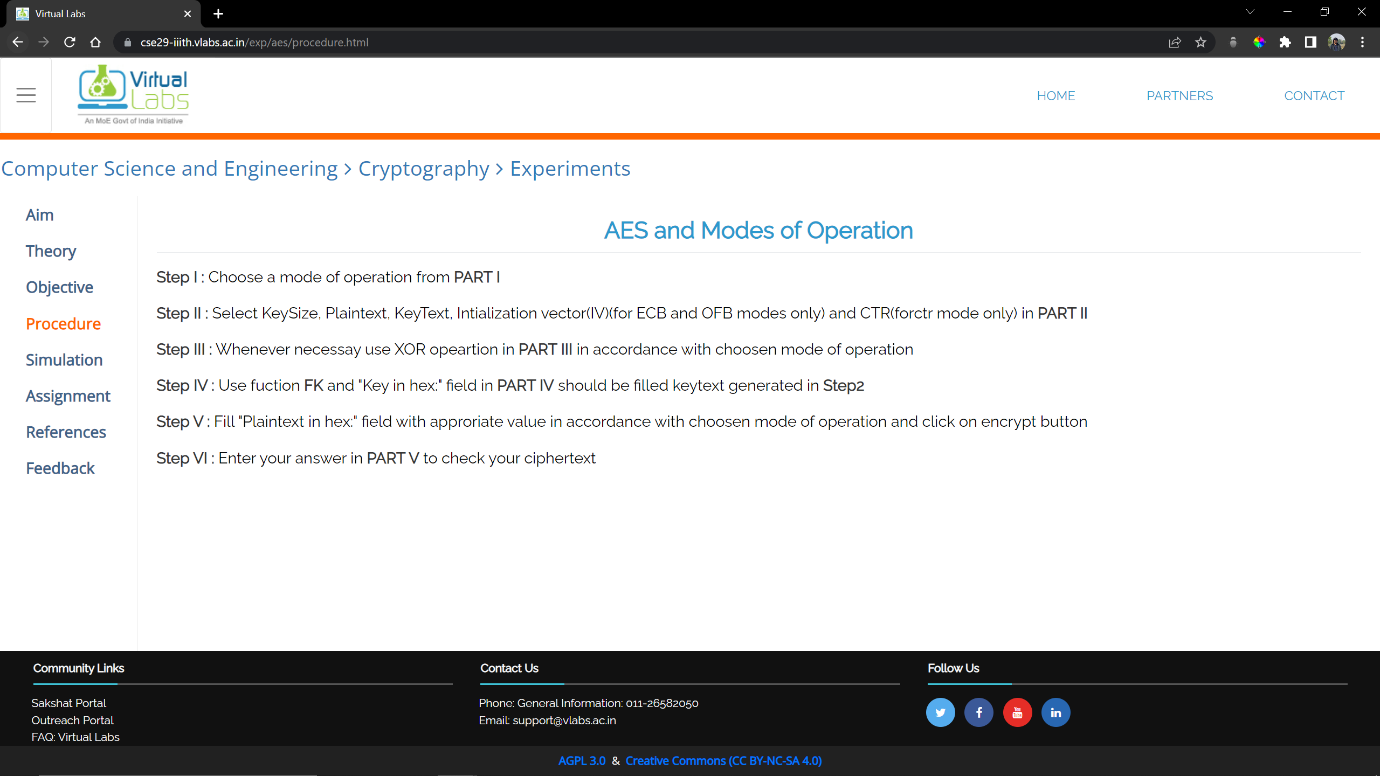
Aim



Theory



Objective

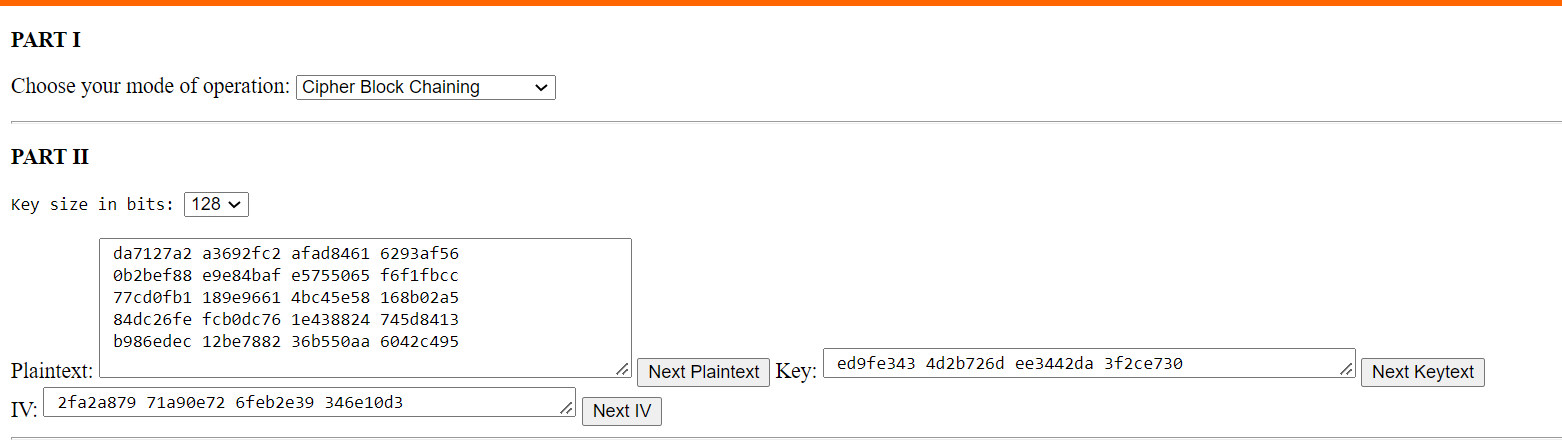


Procedure

We would have to perform in all the 4 modes

Cipher Block Chaining

Generate the values



Plaintext:

da7127a2 a3692fc2 afad8461 6293af56

0b2bef88 e9e84baf e5755065 f6f1fbcc

77cd0fb1 189e9661 4bc45e58 168b02a5

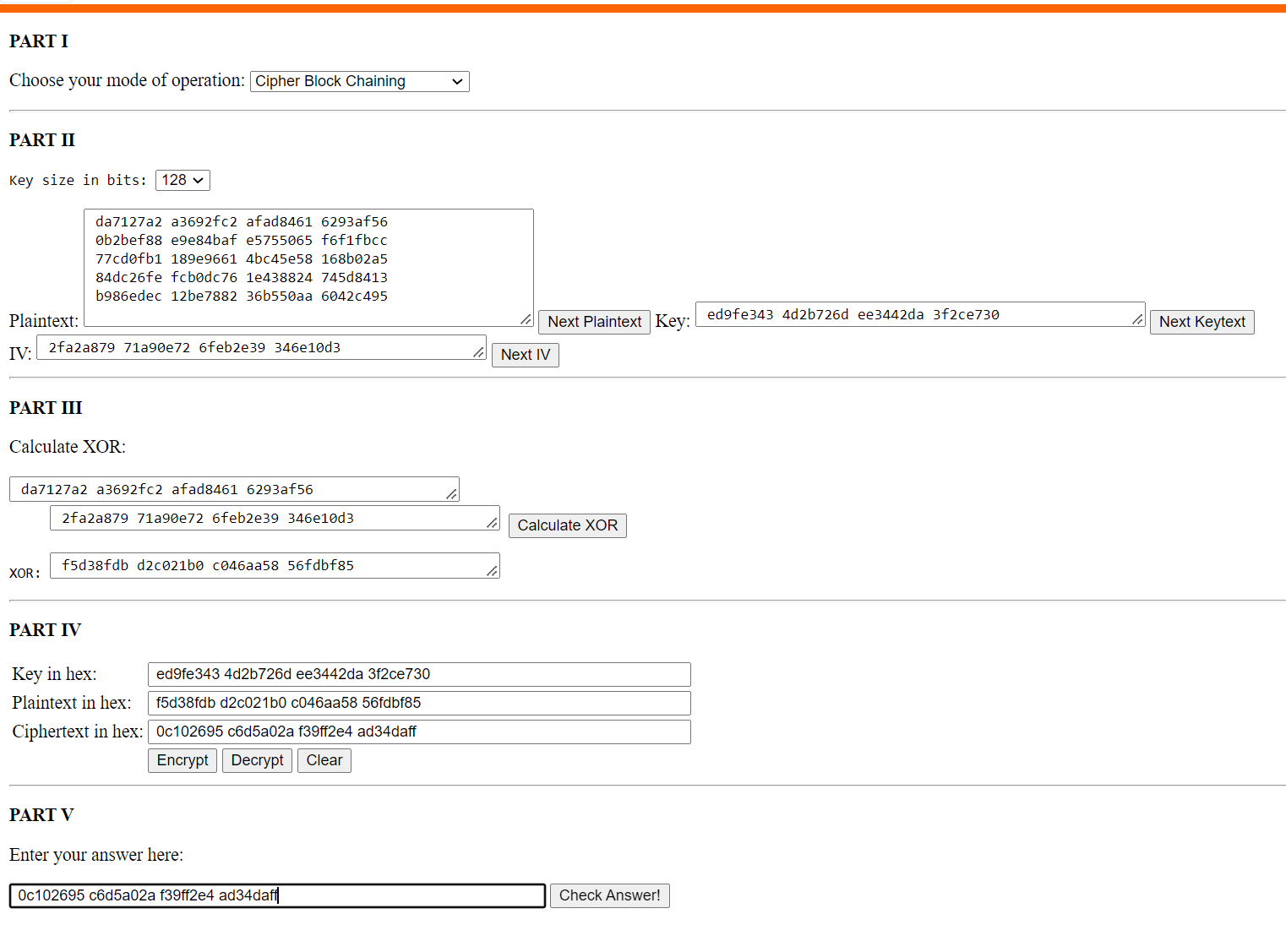
84dc26fe fcb0dc76 1e438824 745d8413

b986edec 12be7882 36b550aa 6042c495

Key: ed9fe343 4d2b726d ee3442da 3f2ce730

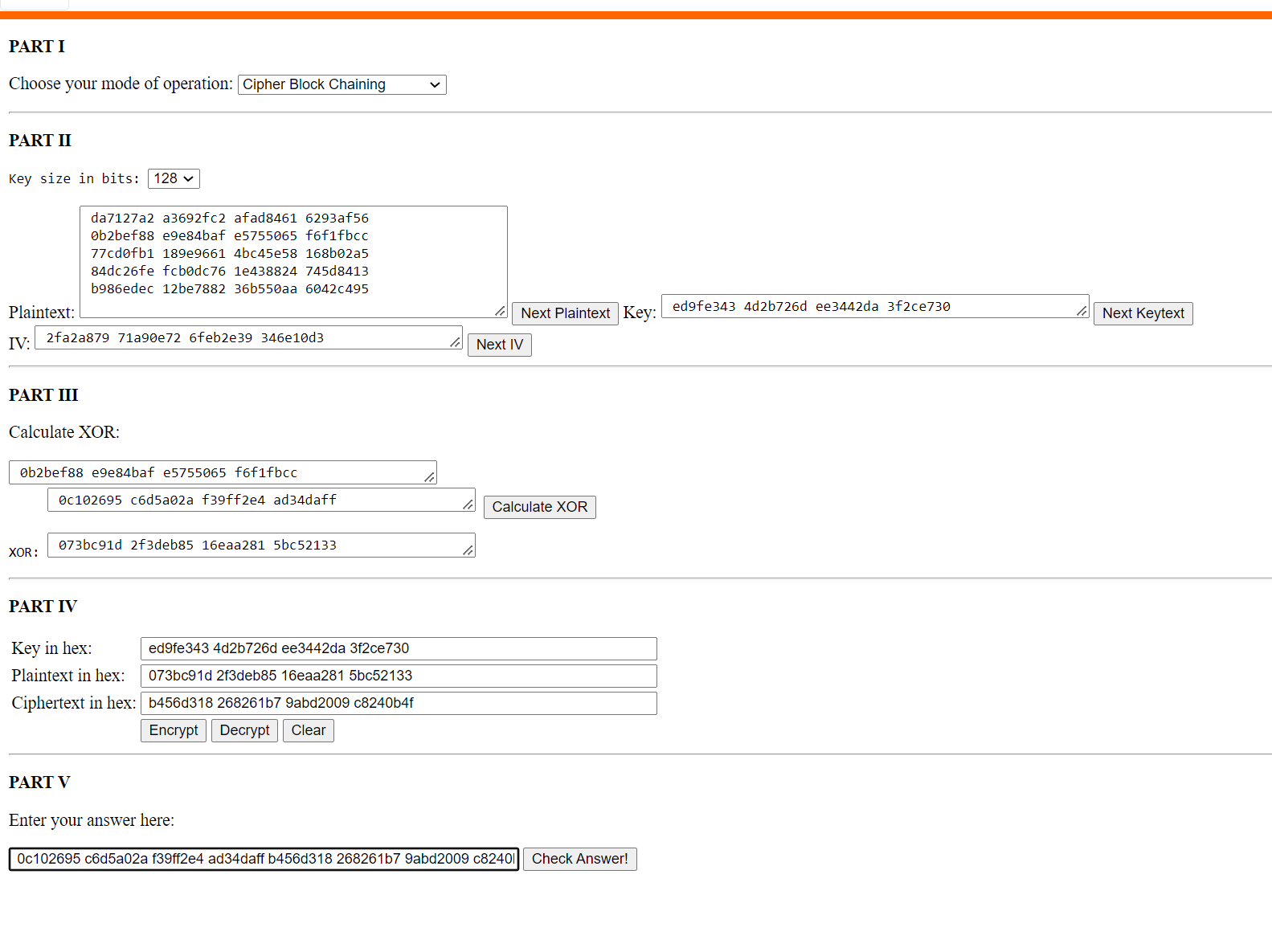
IV: 2fa2a879 71a90e72 6feb2e39 346e10d3

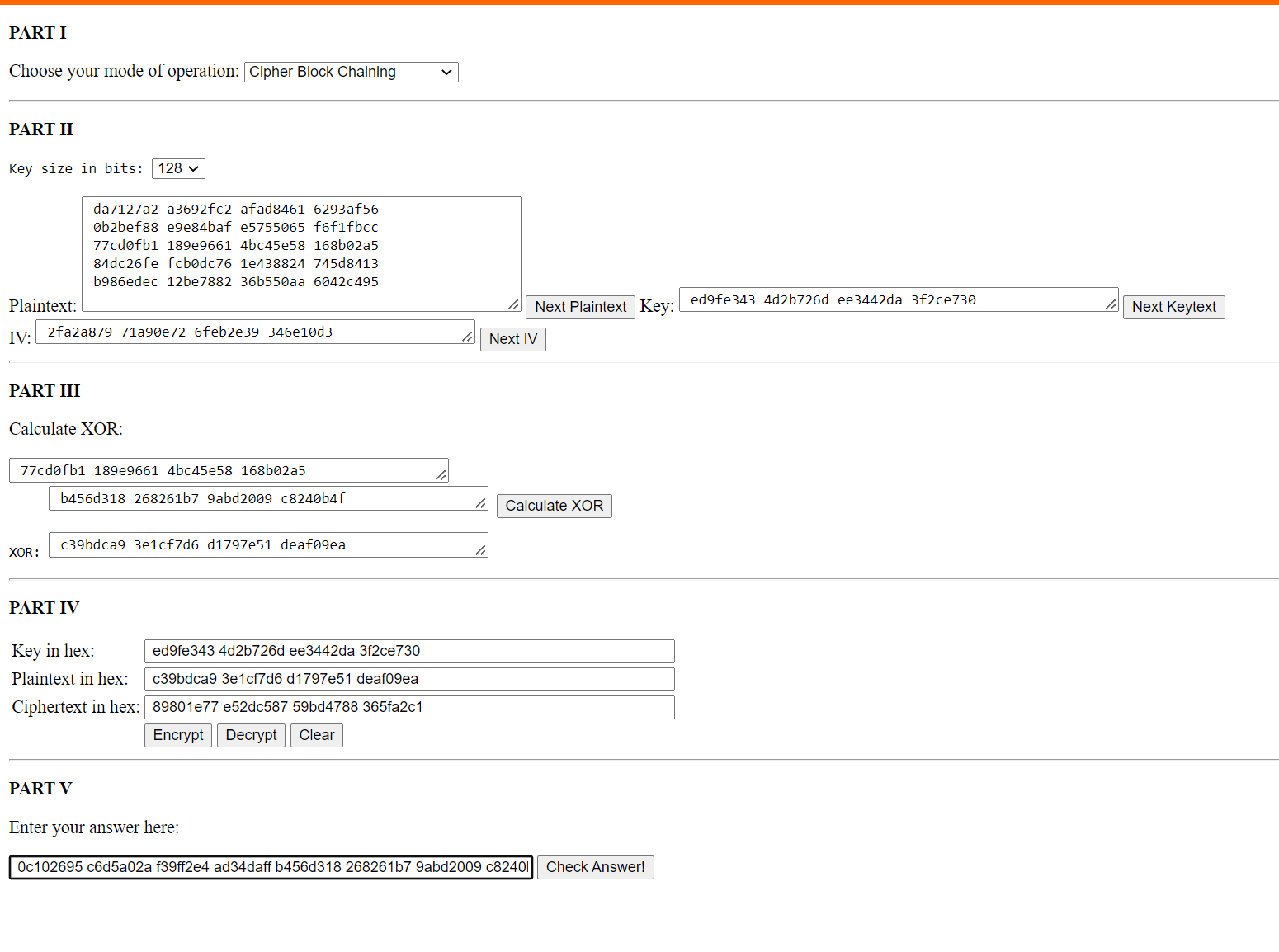
First, take message block 1 and XOR with IV then take that Cipher and Encrypt with the key. And add that result to the final output.

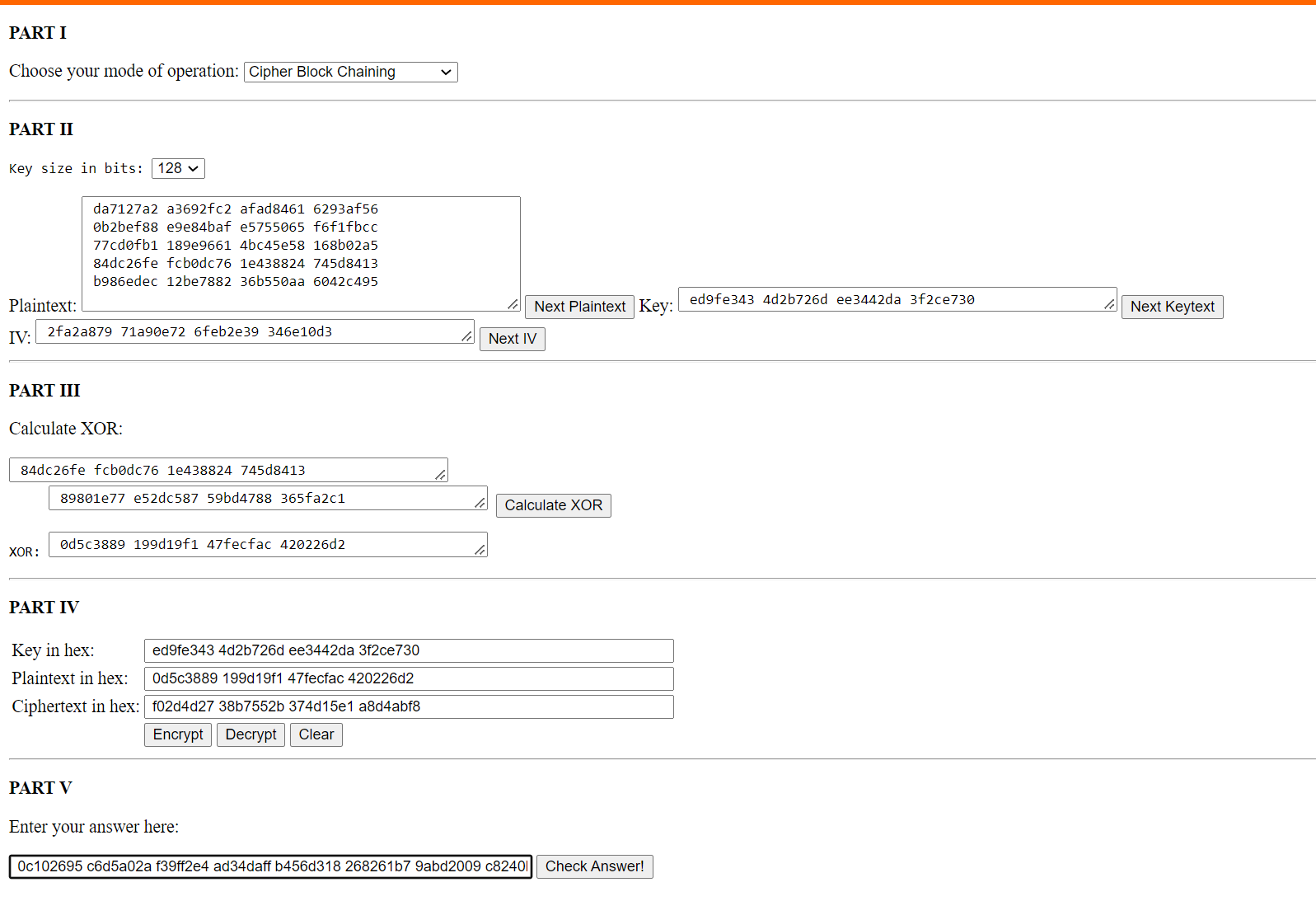


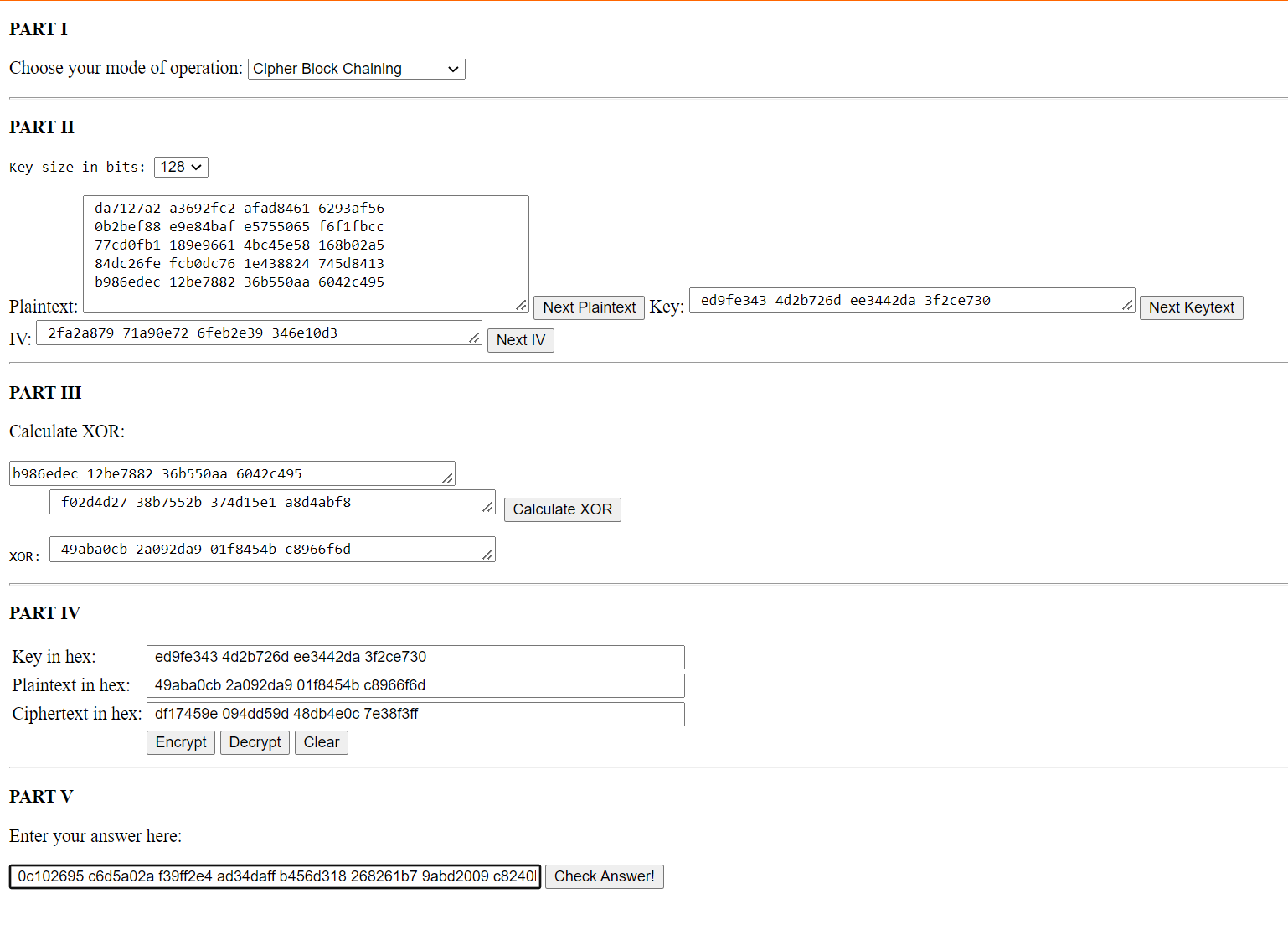
0c102695 c6d5a02a f39ff2e4 ad34daff

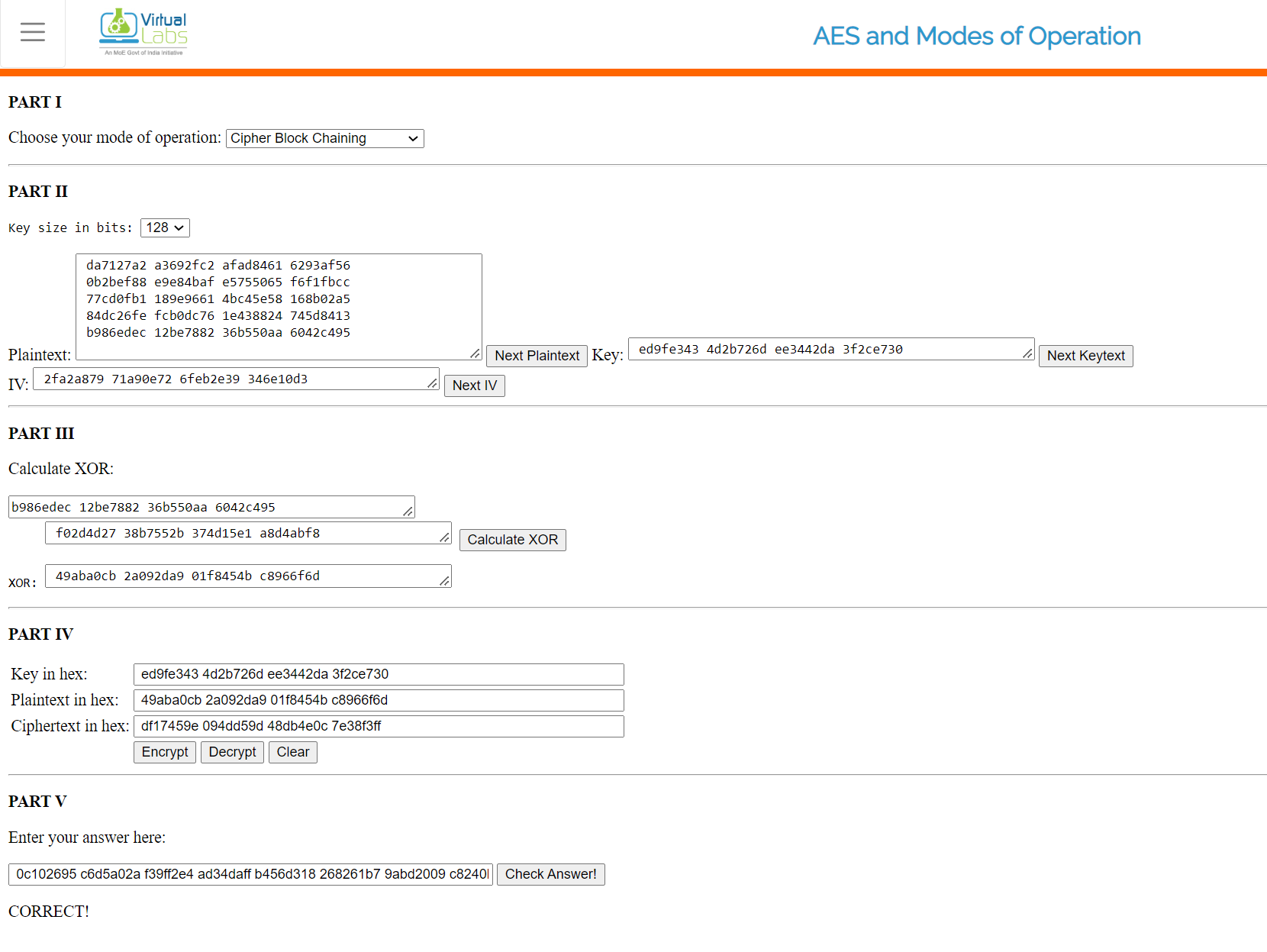
Now, we take the previously generated Cipher and XOR with Plaintext of 2nd part. Then Encrypt with Key and Append to the final output.











We repeat this procedure for all Plaintext

89801e77 e52dc587 59bd4788 365fa2c1  
f02d4d27 38b7552b 374d15e1 a8d4abf8  
df17459e 094dd59d 48db4e0c 7e38f3ff

Encryption: 0c102695 c6d5a02a f39ff2e4 ad34daff b456d318 268261b7 9abd2009 c8240b4f 89801e77 e52dc587 59bd4788 365fa2c1 f02d4d27 38b7552b 374d15e1 a8d4abf8 df17459e 094dd59d 48db4e0c 7e38f3ff

Code:

*import* os

*import* sys

*import* math

*class* AES(object):

*'''AES funtions for a single block*

*'''*

*# Very annoying code:  all is for an object, but no state is kept!*

*# Should just be plain functions in a AES modlule.*

*# valid key sizes*

    keySize = dict(SIZE\_128=16, SIZE\_192=24, SIZE\_256=32)

*# Rijndael S-box*

    sbox =  [*0x*63, *0x*7c, *0x*77, *0x*7b, *0x*f2, *0x*6b, *0x*6f, *0x*c5, *0x*30, *0x*01, *0x*67,

*0x*2b, *0x*fe, *0x*d7, *0x*ab, *0x*76, *0x*ca, *0x*82, *0x*c9, *0x*7d, *0x*fa, *0x*59,

*0x*47, *0x*f0, *0x*ad, *0x*d4, *0x*a2, *0x*af, *0x*9c, *0x*a4, *0x*72, *0x*c0, *0x*b7,

*0x*fd, *0x*93, *0x*26, *0x*36, *0x*3f, *0x*f7, *0x*cc, *0x*34, *0x*a5, *0x*e5, *0x*f1,

*0x*71, *0x*d8, *0x*31, *0x*15, *0x*04, *0x*c7, *0x*23, *0x*c3, *0x*18, *0x*96, *0x*05,

*0x*9a, *0x*07, *0x*12, *0x*80, *0x*e2, *0x*eb, *0x*27, *0x*b2, *0x*75, *0x*09, *0x*83,

*0x*2c, *0x*1a, *0x*1b, *0x*6e, *0x*5a, *0x*a0, *0x*52, *0x*3b, *0x*d6, *0x*b3, *0x*29,

*0x*e3, *0x*2f, *0x*84, *0x*53, *0x*d1, *0x*00, *0x*ed, *0x*20, *0x*fc, *0x*b1, *0x*5b,

*0x*6a, *0x*cb, *0x*be, *0x*39, *0x*4a, *0x*4c, *0x*58, *0x*cf, *0x*d0, *0x*ef, *0x*aa,

*0x*fb, *0x*43, *0x*4d, *0x*33, *0x*85, *0x*45, *0x*f9, *0x*02, *0x*7f, *0x*50, *0x*3c,

*0x*9f, *0x*a8, *0x*51, *0x*a3, *0x*40, *0x*8f, *0x*92, *0x*9d, *0x*38, *0x*f5, *0x*bc,

*0x*b6, *0x*da, *0x*21, *0x*10, *0x*ff, *0x*f3, *0x*d2, *0x*cd, *0x*0c, *0x*13, *0x*ec,

*0x*5f, *0x*97, *0x*44, *0x*17, *0x*c4, *0x*a7, *0x*7e, *0x*3d, *0x*64, *0x*5d, *0x*19,

*0x*73, *0x*60, *0x*81, *0x*4f, *0x*dc, *0x*22, *0x*2a, *0x*90, *0x*88, *0x*46, *0x*ee,

*0x*b8, *0x*14, *0x*de, *0x*5e, *0x*0b, *0x*db, *0x*e0, *0x*32, *0x*3a, *0x*0a, *0x*49,

*0x*06, *0x*24, *0x*5c, *0x*c2, *0x*d3, *0x*ac, *0x*62, *0x*91, *0x*95, *0x*e4, *0x*79,

*0x*e7, *0x*c8, *0x*37, *0x*6d, *0x*8d, *0x*d5, *0x*4e, *0x*a9, *0x*6c, *0x*56, *0x*f4,

*0x*ea, *0x*65, *0x*7a, *0x*ae, *0x*08, *0x*ba, *0x*78, *0x*25, *0x*2e, *0x*1c, *0x*a6,

*0x*b4, *0x*c6, *0x*e8, *0x*dd, *0x*74, *0x*1f, *0x*4b, *0x*bd, *0x*8b, *0x*8a, *0x*70,

*0x*3e, *0x*b5, *0x*66, *0x*48, *0x*03, *0x*f6, *0x*0e, *0x*61, *0x*35, *0x*57, *0x*b9,

*0x*86, *0x*c1, *0x*1d, *0x*9e, *0x*e1, *0x*f8, *0x*98, *0x*11, *0x*69, *0x*d9, *0x*8e,

*0x*94, *0x*9b, *0x*1e, *0x*87, *0x*e9, *0x*ce, *0x*55, *0x*28, *0x*df, *0x*8c, *0x*a1,

*0x*89, *0x*0d, *0x*bf, *0x*e6, *0x*42, *0x*68, *0x*41, *0x*99, *0x*2d, *0x*0f, *0x*b0,

*0x*54, *0x*bb, *0x*16]

*# Rijndael Inverted S-box*

    rsbox = [*0x*52, *0x*09, *0x*6a, *0x*d5, *0x*30, *0x*36, *0x*a5, *0x*38, *0x*bf, *0x*40, *0x*a3,

*0x*9e, *0x*81, *0x*f3, *0x*d7, *0x*fb , *0x*7c, *0x*e3, *0x*39, *0x*82, *0x*9b, *0x*2f,

*0x*ff, *0x*87, *0x*34, *0x*8e, *0x*43, *0x*44, *0x*c4, *0x*de, *0x*e9, *0x*cb , *0x*54,

*0x*7b, *0x*94, *0x*32, *0x*a6, *0x*c2, *0x*23, *0x*3d, *0x*ee, *0x*4c, *0x*95, *0x*0b,

*0x*42, *0x*fa, *0x*c3, *0x*4e , *0x*08, *0x*2e, *0x*a1, *0x*66, *0x*28, *0x*d9, *0x*24,

*0x*b2, *0x*76, *0x*5b, *0x*a2, *0x*49, *0x*6d, *0x*8b, *0x*d1, *0x*25 , *0x*72, *0x*f8,

*0x*f6, *0x*64, *0x*86, *0x*68, *0x*98, *0x*16, *0x*d4, *0x*a4, *0x*5c, *0x*cc, *0x*5d,

*0x*65, *0x*b6, *0x*92 , *0x*6c, *0x*70, *0x*48, *0x*50, *0x*fd, *0x*ed, *0x*b9, *0x*da,

*0x*5e, *0x*15, *0x*46, *0x*57, *0x*a7, *0x*8d, *0x*9d, *0x*84 , *0x*90, *0x*d8, *0x*ab,

*0x*00, *0x*8c, *0x*bc, *0x*d3, *0x*0a, *0x*f7, *0x*e4, *0x*58, *0x*05, *0x*b8, *0x*b3,

*0x*45, *0x*06 , *0x*d0, *0x*2c, *0x*1e, *0x*8f, *0x*ca, *0x*3f, *0x*0f, *0x*02, *0x*c1,

*0x*af, *0x*bd, *0x*03, *0x*01, *0x*13, *0x*8a, *0x*6b , *0x*3a, *0x*91, *0x*11, *0x*41,

*0x*4f, *0x*67, *0x*dc, *0x*ea, *0x*97, *0x*f2, *0x*cf, *0x*ce, *0x*f0, *0x*b4, *0x*e6,

*0x*73 , *0x*96, *0x*ac, *0x*74, *0x*22, *0x*e7, *0x*ad, *0x*35, *0x*85, *0x*e2, *0x*f9,

*0x*37, *0x*e8, *0x*1c, *0x*75, *0x*df, *0x*6e , *0x*47, *0x*f1, *0x*1a, *0x*71, *0x*1d,

*0x*29, *0x*c5, *0x*89, *0x*6f, *0x*b7, *0x*62, *0x*0e, *0x*aa, *0x*18, *0x*be, *0x*1b ,

*0x*fc, *0x*56, *0x*3e, *0x*4b, *0x*c6, *0x*d2, *0x*79, *0x*20, *0x*9a, *0x*db, *0x*c0,

*0x*fe, *0x*78, *0x*cd, *0x*5a, *0x*f4 , *0x*1f, *0x*dd, *0x*a8, *0x*33, *0x*88, *0x*07,

*0x*c7, *0x*31, *0x*b1, *0x*12, *0x*10, *0x*59, *0x*27, *0x*80, *0x*ec, *0x*5f , *0x*60,

*0x*51, *0x*7f, *0x*a9, *0x*19, *0x*b5, *0x*4a, *0x*0d, *0x*2d, *0x*e5, *0x*7a, *0x*9f,

*0x*93, *0x*c9, *0x*9c, *0x*ef , *0x*a0, *0x*e0, *0x*3b, *0x*4d, *0x*ae, *0x*2a, *0x*f5,

*0x*b0, *0x*c8, *0x*eb, *0x*bb, *0x*3c, *0x*83, *0x*53, *0x*99, *0x*61 , *0x*17, *0x*2b,

*0x*04, *0x*7e, *0x*ba, *0x*77, *0x*d6, *0x*26, *0x*e1, *0x*69, *0x*14, *0x*63, *0x*55,

*0x*21, *0x*0c, *0x*7d]

*def* getSBoxValue(self,num):

*"""Retrieves a given S-Box Value"""*

*return* *self*.sbox[num]

*def* getSBoxInvert(self,num):

*"""Retrieves a given Inverted S-Box Value"""*

*return* *self*.rsbox[num]

*def* rotate(self, word):

*""" Rijndael's key schedule rotate operation.*

*Rotate a word eight bits to the left: eg, rotate(1d2c3a4f) == 2c3a4f1d*

*Word is an char list of size 4 (32 bits overall).*

*"""*

*return* word[1:] + word[:1]

*# Rijndael Rcon*

    Rcon = [*0x*8d, *0x*01, *0x*02, *0x*04, *0x*08, *0x*10, *0x*20, *0x*40, *0x*80, *0x*1b, *0x*36,

*0x*6c, *0x*d8, *0x*ab, *0x*4d, *0x*9a, *0x*2f, *0x*5e, *0x*bc, *0x*63, *0x*c6, *0x*97,

*0x*35, *0x*6a, *0x*d4, *0x*b3, *0x*7d, *0x*fa, *0x*ef, *0x*c5, *0x*91, *0x*39, *0x*72,

*0x*e4, *0x*d3, *0x*bd, *0x*61, *0x*c2, *0x*9f, *0x*25, *0x*4a, *0x*94, *0x*33, *0x*66,

*0x*cc, *0x*83, *0x*1d, *0x*3a, *0x*74, *0x*e8, *0x*cb, *0x*8d, *0x*01, *0x*02, *0x*04,

*0x*08, *0x*10, *0x*20, *0x*40, *0x*80, *0x*1b, *0x*36, *0x*6c, *0x*d8, *0x*ab, *0x*4d,

*0x*9a, *0x*2f, *0x*5e, *0x*bc, *0x*63, *0x*c6, *0x*97, *0x*35, *0x*6a, *0x*d4, *0x*b3,

*0x*7d, *0x*fa, *0x*ef, *0x*c5, *0x*91, *0x*39, *0x*72, *0x*e4, *0x*d3, *0x*bd, *0x*61,

*0x*c2, *0x*9f, *0x*25, *0x*4a, *0x*94, *0x*33, *0x*66, *0x*cc, *0x*83, *0x*1d, *0x*3a,

*0x*74, *0x*e8, *0x*cb, *0x*8d, *0x*01, *0x*02, *0x*04, *0x*08, *0x*10, *0x*20, *0x*40,

*0x*80, *0x*1b, *0x*36, *0x*6c, *0x*d8, *0x*ab, *0x*4d, *0x*9a, *0x*2f, *0x*5e, *0x*bc,

*0x*63, *0x*c6, *0x*97, *0x*35, *0x*6a, *0x*d4, *0x*b3, *0x*7d, *0x*fa, *0x*ef, *0x*c5,

*0x*91, *0x*39, *0x*72, *0x*e4, *0x*d3, *0x*bd, *0x*61, *0x*c2, *0x*9f, *0x*25, *0x*4a,

*0x*94, *0x*33, *0x*66, *0x*cc, *0x*83, *0x*1d, *0x*3a, *0x*74, *0x*e8, *0x*cb, *0x*8d,

*0x*01, *0x*02, *0x*04, *0x*08, *0x*10, *0x*20, *0x*40, *0x*80, *0x*1b, *0x*36, *0x*6c,

*0x*d8, *0x*ab, *0x*4d, *0x*9a, *0x*2f, *0x*5e, *0x*bc, *0x*63, *0x*c6, *0x*97, *0x*35,

*0x*6a, *0x*d4, *0x*b3, *0x*7d, *0x*fa, *0x*ef, *0x*c5, *0x*91, *0x*39, *0x*72, *0x*e4,

*0x*d3, *0x*bd, *0x*61, *0x*c2, *0x*9f, *0x*25, *0x*4a, *0x*94, *0x*33, *0x*66, *0x*cc,

*0x*83, *0x*1d, *0x*3a, *0x*74, *0x*e8, *0x*cb, *0x*8d, *0x*01, *0x*02, *0x*04, *0x*08,

*0x*10, *0x*20, *0x*40, *0x*80, *0x*1b, *0x*36, *0x*6c, *0x*d8, *0x*ab, *0x*4d, *0x*9a,

*0x*2f, *0x*5e, *0x*bc, *0x*63, *0x*c6, *0x*97, *0x*35, *0x*6a, *0x*d4, *0x*b3, *0x*7d,

*0x*fa, *0x*ef, *0x*c5, *0x*91, *0x*39, *0x*72, *0x*e4, *0x*d3, *0x*bd, *0x*61, *0x*c2,

*0x*9f, *0x*25, *0x*4a, *0x*94, *0x*33, *0x*66, *0x*cc, *0x*83, *0x*1d, *0x*3a, *0x*74,

*0x*e8, *0x*cb ]

*def* getRconValue(self, num):

*"""Retrieves a given Rcon Value"""*

*return* *self*.Rcon[num]

*def* core(self, word, iteration):

*"""Key schedule core."""*

*# rotate the 32-bit word 8 bits to the left*

        word = *self*.rotate(word)

*# apply S-Box substitution on all 4 parts of the 32-bit word*

*for* i *in* range(4):

            word[i] = *self*.getSBoxValue(word[i])

*# XOR the output of the rcon operation with i to the first part*

*# (leftmost) only*

        word[0] = word[0] ^ *self*.getRconValue(iteration)

*return* word

*def* expandKey(self, key, size, expandedKeySize):

*"""Rijndael's key expansion.*

*Expands an 128,192,256 key into an 176,208,240 bytes key*

*expandedKey is a char list of large enough size,*

*key is the non-expanded key.*

*"""*

*# current expanded keySize, in bytes*

        currentSize = 0

        rconIteration = 1

        expandedKey = [0] \* expandedKeySize

*# set the 16, 24, 32 bytes of the expanded key to the input key*

*for* j *in* range(size):

            expandedKey[j] = key[j]

        currentSize += size

*while* currentSize < expandedKeySize:

*# assign the previous 4 bytes to the temporary value t*

            t = expandedKey[currentSize-4:currentSize]

*# every 16,24,32 bytes we apply the core schedule to t*

*# and increment rconIteration afterwards*

*if* currentSize % size == 0:

                t = *self*.core(t, rconIteration)

                rconIteration += 1

*# For 256-bit keys, we add an extra sbox to the calculation*

*if* size == *self*.keySize["SIZE\_256"] and ((currentSize % size) == 16):

*for* l *in* range(4): t[l] = *self*.getSBoxValue(t[l])

*# We XOR t with the four-byte block 16,24,32 bytes before the new*

*# expanded key.  This becomes the next four bytes in the expanded*

*# key.*

*for* m *in* range(4):

                expandedKey[currentSize] = expandedKey[currentSize - size] ^ \

                        t[m]

                currentSize += 1

*return* expandedKey

*def* addRoundKey(self, state, roundKey):

*"""Adds (XORs) the round key to the state."""*

*for* i *in* range(16):

            state[i] ^= roundKey[i]

*return* state

*def* createRoundKey(self, expandedKey, roundKeyPointer):

*"""Create a round key.*

*Creates a round key from the given expanded key and the*

*position within the expanded key.*

*"""*

        roundKey = [0] \* 16

*for* i *in* range(4):

*for* j *in* range(4):

                roundKey[j\*4+i] = expandedKey[roundKeyPointer + i\*4 + j]

*return* roundKey

*def* galois\_multiplication(self, a, b):

*"""Galois multiplication of 8 bit characters a and b."""*

        p = 0

*for* counter *in* range(8):

*if* b & 1: p ^= a

            hi\_bit\_set = a & *0x*80

            a <<= 1

*# keep a 8 bit*

            a &= *0x*FF

*if* hi\_bit\_set:

                a ^= *0x*1b

            b >>= 1

*return* p

*#*

*# substitute all the values from the state with the value in the SBox*

*# using the state value as index for the SBox*

*#*

*def* subBytes(self, state, isInv):

*if* isInv: getter = *self*.getSBoxInvert

*else*: getter = *self*.getSBoxValue

*for* i *in* range(16): state[i] = getter(state[i])

*return* state

*# iterate over the 4 rows and call shiftRow() with that row*

*def* shiftRows(self, state, isInv):

*for* i *in* range(4):

            state = *self*.shiftRow(state, i\*4, i, isInv)

*return* state

*# each iteration shifts the row to the left by 1*

*def* shiftRow(self, state, statePointer, nbr, isInv):

*for* i *in* range(nbr):

*if* isInv:

                state[statePointer:statePointer+4] = \

                        state[statePointer+3:statePointer+4] + \

                        state[statePointer:statePointer+3]

*else*:

                state[statePointer:statePointer+4] = \

                        state[statePointer+1:statePointer+4] + \

                        state[statePointer:statePointer+1]

*return* state

*# galois multiplication of the 4x4 matrix*

*def* mixColumns(self, state, isInv):

*# iterate over the 4 columns*

*for* i *in* range(4):

*# construct one column by slicing over the 4 rows*

            column = state[i:i+16:4]

*# apply the mixColumn on one column*

            column = *self*.mixColumn(column, isInv)

*# put the values back into the state*

            state[i:i+16:4] = column

*return* state

*# galois multiplication of 1 column of the 4x4 matrix*

*def* mixColumn(self, column, isInv):

*if* isInv: mult = [14, 9, 13, 11]

*else*: mult = [2, 1, 1, 3]

        cpy = list(column)

        g = *self*.galois\_multiplication

        column[0] = g(cpy[0], mult[0]) ^ g(cpy[3], mult[1]) ^ \

                    g(cpy[2], mult[2]) ^ g(cpy[1], mult[3])

        column[1] = g(cpy[1], mult[0]) ^ g(cpy[0], mult[1]) ^ \

                    g(cpy[3], mult[2]) ^ g(cpy[2], mult[3])

        column[2] = g(cpy[2], mult[0]) ^ g(cpy[1], mult[1]) ^ \

                    g(cpy[0], mult[2]) ^ g(cpy[3], mult[3])

        column[3] = g(cpy[3], mult[0]) ^ g(cpy[2], mult[1]) ^ \

                    g(cpy[1], mult[2]) ^ g(cpy[0], mult[3])

*return* column

*# applies the 4 operations of the forward round in sequence*

*def* aes\_round(self, state, roundKey):

        state = *self*.subBytes(state, False)

        state = *self*.shiftRows(state, False)

        state = *self*.mixColumns(state, False)

        state = *self*.addRoundKey(state, roundKey)

*return* state

*# applies the 4 operations of the inverse round in sequence*

*def* aes\_invRound(self, state, roundKey):

        state = *self*.shiftRows(state, True)

        state = *self*.subBytes(state, True)

        state = *self*.addRoundKey(state, roundKey)

        state = *self*.mixColumns(state, True)

*return* state

*# Perform the initial operations, the standard round, and the final*

*# operations of the forward aes, creating a round key for each round*

*def* aes\_main(self, state, expandedKey, nbrRounds):

        state = *self*.addRoundKey(state, *self*.createRoundKey(expandedKey, 0))

        i = 1

*while* i < nbrRounds:

            state = *self*.aes\_round(state,

*self*.createRoundKey(expandedKey, 16\*i))

            i += 1

        state = *self*.subBytes(state, False)

        state = *self*.shiftRows(state, False)

        state = *self*.addRoundKey(state,

*self*.createRoundKey(expandedKey, 16\*nbrRounds))

*return* state

*# Perform the initial operations, the standard round, and the final*

*# operations of the inverse aes, creating a round key for each round*

*def* aes\_invMain(self, state, expandedKey, nbrRounds):

        state = *self*.addRoundKey(state,

*self*.createRoundKey(expandedKey, 16\*nbrRounds))

        i = nbrRounds - 1

*while* i > 0:

            state = *self*.aes\_invRound(state,

*self*.createRoundKey(expandedKey, 16\*i))

            i -= 1

        state = *self*.shiftRows(state, True)

        state = *self*.subBytes(state, True)

        state = *self*.addRoundKey(state, *self*.createRoundKey(expandedKey, 0))

*return* state

*# encrypts a 128 bit input block against the given key of size specified*

*def* encrypt(self, iput, key, size):

        output = [0] \* 16

*# the number of rounds*

        nbrRounds = 0

*# the 128 bit block to encode*

        block = [0] \* 16

*# set the number of rounds*

*if* size == *self*.keySize["SIZE\_128"]: nbrRounds = 10

*elif* size == *self*.keySize["SIZE\_192"]: nbrRounds = 12

*elif* size == *self*.keySize["SIZE\_256"]: nbrRounds = 14

*else*: *return* None

*# the expanded keySize*

        expandedKeySize = 16\*(nbrRounds+1)

*# Set the block values, for the block:*

*# a0,0 a0,1 a0,2 a0,3*

*# a1,0 a1,1 a1,2 a1,3*

*# a2,0 a2,1 a2,2 a2,3*

*# a3,0 a3,1 a3,2 a3,3*

*# the mapping order is a0,0 a1,0 a2,0 a3,0 a0,1 a1,1 ... a2,3 a3,3*

*#*

*# iterate over the columns*

*for* i *in* range(4):

*# iterate over the rows*

*for* j *in* range(4):

                block[(i+(j\*4))] = iput[(i\*4)+j]

*# expand the key into an 176, 208, 240 bytes key*

*# the expanded key*

        expandedKey = *self*.expandKey(key, size, expandedKeySize)

*# encrypt the block using the expandedKey*

        block = *self*.aes\_main(block, expandedKey, nbrRounds)

*# unmap the block again into the output*

*for* k *in* range(4):

*# iterate over the rows*

*for* l *in* range(4):

                output[(k\*4)+l] = block[(k+(l\*4))]

*return* output

*# decrypts a 128 bit input block against the given key of size specified*

*def* decrypt(self, iput, key, size):

        output = [0] \* 16

*# the number of rounds*

        nbrRounds = 0

*# the 128 bit block to decode*

        block = [0] \* 16

*# set the number of rounds*

*if* size == *self*.keySize["SIZE\_128"]: nbrRounds = 10

*elif* size == *self*.keySize["SIZE\_192"]: nbrRounds = 12

*elif* size == *self*.keySize["SIZE\_256"]: nbrRounds = 14

*else*: *return* None

*# the expanded keySize*

        expandedKeySize = 16\*(nbrRounds+1)

*# Set the block values, for the block:*

*# a0,0 a0,1 a0,2 a0,3*

*# a1,0 a1,1 a1,2 a1,3*

*# a2,0 a2,1 a2,2 a2,3*

*# a3,0 a3,1 a3,2 a3,3*

*# the mapping order is a0,0 a1,0 a2,0 a3,0 a0,1 a1,1 ... a2,3 a3,3*

*# iterate over the columns*

*for* i *in* range(4):

*# iterate over the rows*

*for* j *in* range(4):

                block[(i+(j\*4))] = iput[(i\*4)+j]

*# expand the key into an 176, 208, 240 bytes key*

        expandedKey = *self*.expandKey(key, size, expandedKeySize)

*# decrypt the block using the expandedKey*

        block = *self*.aes\_invMain(block, expandedKey, nbrRounds)

*# unmap the block again into the output*

*for* k *in* range(4):

*# iterate over the rows*

*for* l *in* range(4):

                output[(k\*4)+l] = block[(k+(l\*4))]

*return* output

*class* AESModeOfOperation(object):

*'''Handles AES with plaintext consistingof multiple blocks.*

*Choice of block encoding modes:  OFT, CFB, CBC*

*'''*

*# Very annoying code:  all is for an object, but no state is kept!*

*# Should just be plain functions in an AES\_BlockMode module.*

    aes = AES()

*# structure of supported modes of operation*

    modeOfOperation = dict(OFB=0, CFB=1, CBC=2)

*# converts a 16 character string into a number array*

*def* convertString(self, string, start, end, mode):

*if* end - start > 16: end = start + 16

*if* mode == *self*.modeOfOperation["CBC"]: ar = [0] \* 16

*else*: ar = []

        i = start

        j = 0

*while* len(ar) < end - start:

            ar.append(0)

*while* i < end:

            ar[j] = ord(string[i])

            j += 1

            i += 1

*return* ar

*# Mode of Operation Encryption*

*# stringIn - Input String*

*# mode - mode of type modeOfOperation*

*# hexKey - a hex key of the bit length size*

*# size - the bit length of the key*

*# hexIV - the 128 bit hex Initilization Vector*

*def* encrypt(self, stringIn, mode, key, size, IV):

*if* len(key) % size:

*return* None

*if* len(IV) % 16:

*return* None

*# the AES input/output*

        plaintext = []

        iput = [0] \* 16

        output = []

        ciphertext = [0] \* 16

*# the output cipher string*

        cipherOut = []

*# char firstRound*

        firstRound = True

*if* stringIn != None:

*for* j *in* range(int(math.ceil(float(len(stringIn))/16))):

                start = j\*16

                end = j\*16+16

*if*  end > len(stringIn):

                    end = len(stringIn)

                plaintext = *self*.convertString(stringIn, start, end, mode)

*# print 'PT@%s:%s' % (j, plaintext)*

*if* mode == *self*.modeOfOperation["CFB"]:

*if* firstRound:

                        output = *self*.aes.encrypt(IV, key, size)

                        firstRound = False

*else*:

                        output = *self*.aes.encrypt(iput, key, size)

*for* i *in* range(16):

*if* len(plaintext)-1 < i:

                            ciphertext[i] = 0 ^ output[i]

*elif* len(output)-1 < i:

                            ciphertext[i] = plaintext[i] ^ 0

*elif* len(plaintext)-1 < i and len(output) < i:

                            ciphertext[i] = 0 ^ 0

*else*:

                            ciphertext[i] = plaintext[i] ^ output[i]

*for* k *in* range(end-start):

                        cipherOut.append(ciphertext[k])

                    iput = ciphertext

*elif* mode == *self*.modeOfOperation["OFB"]:

*if* firstRound:

                        output = *self*.aes.encrypt(IV, key, size)

                        firstRound = False

*else*:

                        output = *self*.aes.encrypt(iput, key, size)

*for* i *in* range(16):

*if* len(plaintext)-1 < i:

                            ciphertext[i] = 0 ^ output[i]

*elif* len(output)-1 < i:

                            ciphertext[i] = plaintext[i] ^ 0

*elif* len(plaintext)-1 < i and len(output) < i:

                            ciphertext[i] = 0 ^ 0

*else*:

                            ciphertext[i] = plaintext[i] ^ output[i]

*for* k *in* range(end-start):

                        cipherOut.append(ciphertext[k])

                    iput = output

*elif* mode == *self*.modeOfOperation["CBC"]:

*for* i *in* range(16):

*if* firstRound:

                            iput[i] =  plaintext[i] ^ IV[i]

*else*:

                            iput[i] =  plaintext[i] ^ ciphertext[i]

*# print 'IP@%s:%s' % (j, iput)*

                    firstRound = False

                    ciphertext = *self*.aes.encrypt(iput, key, size)

*# always 16 bytes because of the padding for CBC*

*for* k *in* range(16):

                        cipherOut.append(ciphertext[k])

*return* mode, len(stringIn), cipherOut

*# Mode of Operation Decryption*

*# cipherIn - Encrypted String*

*# originalsize - The unencrypted string length - required for CBC*

*# mode - mode of type modeOfOperation*

*# key - a number array of the bit length size*

*# size - the bit length of the key*

*# IV - the 128 bit number array Initilization Vector*

*def* decrypt(self, cipherIn, originalsize, mode, key, size, IV):

*# cipherIn = unescCtrlChars(cipherIn)*

*if* len(key) % size:

*return* None

*if* len(IV) % 16:

*return* None

*# the AES input/output*

        ciphertext = []

        iput = []

        output = []

        plaintext = [0] \* 16

*# the output plain text character list*

        chrOut = []

*# char firstRound*

        firstRound = True

*if* cipherIn != None:

*for* j *in* range(int(math.ceil(float(len(cipherIn))/16))):

                start = j\*16

                end = j\*16+16

*if* j\*16+16 > len(cipherIn):

                    end = len(cipherIn)

                ciphertext = cipherIn[start:end]

*if* mode == *self*.modeOfOperation["CFB"]:

*if* firstRound:

                        output = *self*.aes.encrypt(IV, key, size)

                        firstRound = False

*else*:

                        output = *self*.aes.encrypt(iput, key, size)

*for* i *in* range(16):

*if* len(output)-1 < i:

                            plaintext[i] = 0 ^ ciphertext[i]

*elif* len(ciphertext)-1 < i:

                            plaintext[i] = output[i] ^ 0

*elif* len(output)-1 < i and len(ciphertext) < i:

                            plaintext[i] = 0 ^ 0

*else*:

                            plaintext[i] = output[i] ^ ciphertext[i]

*for* k *in* range(end-start):

                        chrOut.append(chr(plaintext[k]))

                    iput = ciphertext

*elif* mode == *self*.modeOfOperation["OFB"]:

*if* firstRound:

                        output = *self*.aes.encrypt(IV, key, size)

                        firstRound = False

*else*:

                        output = *self*.aes.encrypt(iput, key, size)

*for* i *in* range(16):

*if* len(output)-1 < i:

                            plaintext[i] = 0 ^ ciphertext[i]

*elif* len(ciphertext)-1 < i:

                            plaintext[i] = output[i] ^ 0

*elif* len(output)-1 < i and len(ciphertext) < i:

                            plaintext[i] = 0 ^ 0

*else*:

                            plaintext[i] = output[i] ^ ciphertext[i]

*for* k *in* range(end-start):

                        chrOut.append(chr(plaintext[k]))

                    iput = output

*elif* mode == *self*.modeOfOperation["CBC"]:

                    output = *self*.aes.decrypt(ciphertext, key, size)

*for* i *in* range(16):

*if* firstRound:

                            plaintext[i] = IV[i] ^ output[i]

*else*:

                            plaintext[i] = iput[i] ^ output[i]

                    firstRound = False

*if* originalsize is not None and originalsize < end:

*for* k *in* range(originalsize-start):

                            chrOut.append(chr(plaintext[k]))

*else*:

*for* k *in* range(end-start):

                            chrOut.append(chr(plaintext[k]))

                    iput = ciphertext

*return* "".join(chrOut)

*def* append\_PKCS7\_padding(s):

*"""return s padded to a multiple of 16-bytes by PKCS7 padding"""*

    numpads = 16 - (len(s)%16)

*return* s + numpads\*chr(numpads)

*def* strip\_PKCS7\_padding(s):

*"""return s stripped of PKCS7 padding"""*

*if* len(s)%16 or not s:

*raise* ValueError("String of len %d can't be PCKS7-padded" % len(s))

    numpads = ord(s[-1])

*if* numpads > 16:

*raise* ValueError("String ending with %r can't be PCKS7-padded" % s[-1])

*return* s[:-numpads]

*def* encryptData(key, data, mode=AESModeOfOperation.modeOfOperation["CBC"]):

*"""encrypt `data` using `key`*

*`key` should be a string of bytes.*

*returned cipher is a string of bytes prepended with the initialization*

*vector.*

*"""*

    key = list(key)

*if* mode == AESModeOfOperation.modeOfOperation["CBC"]:

        data = append\_PKCS7\_padding(data)

    keysize = len(key)

*assert* keysize in list(AES.keySize.values()), 'invalid key size: %s' % keysize

*# create a new iv using random data*

    iv = [i *for* i *in* os.urandom(16)]

    moo = AESModeOfOperation()

    (mode, length, ciph) = moo.encrypt(data, mode, key, keysize, iv)

*# With padding, the original length does not need to be known. It's a bad*

*# idea to store the original message length.*

*# prepend the iv.*

*return* ''.join(map(chr, iv)) + ''.join(map(chr, ciph))

*def* decryptData(key, data, mode=AESModeOfOperation.modeOfOperation["CBC"]):

*"""decrypt `data` using `key`*

*`key` should be a string of bytes.*

*`data` should have the initialization vector prepended as a string of*

*ordinal values.*

*"""*

    key = list(key)

    keysize = len(key)

*assert* keysize in list(AES.keySize.values()), 'invalid key size: %s' % keysize

*# iv is first 16 bytes*

    iv = list(map(ord, data[:16]))

    data = list(map(ord, data[16:]))

    moo = AESModeOfOperation()

    decr = moo.decrypt(data, None, mode, key, keysize, iv)

*if* mode == AESModeOfOperation.modeOfOperation["CBC"]:

        decr = strip\_PKCS7\_padding(decr)

*return* decr

*def* generateRandomKey(keysize):

*"""Generates a key from random data of length `keysize`.*

*The returned key is a string of bytes.*

*"""*

*if* keysize not in (16, 24, 32):

        emsg = 'Invalid keysize, %s. Should be one of (16, 24, 32).'

*raise* ValueError(emsg % keysize)

*return* os.urandom(keysize)

*def* testStr(cleartext, keysize=16, modeName = "CBC"):

*'''Test with random key, choice of mode.'''*

    print('Random key test', 'Mode:', modeName)

    print('cleartext:', cleartext)

    key =  generateRandomKey(keysize)

    print('Key:', [x *for* x *in* key])

    mode = AESModeOfOperation.modeOfOperation[modeName]

    cipher = encryptData(key, cleartext, mode)

    print('Cipher:', [ord(x) *for* x *in* cipher])

    decr = decryptData(key, cipher, mode)

    print('Decrypted:', decr)

*def* convertToList(keyString):

    lst = []

*for* k *in* keyString:

        lst.append(ord(k))

*return* lst

*if* \_\_name\_\_ == "\_\_main\_\_":

    moo = AESModeOfOperation()

    print("-- AES Encryption and Decryption --")

    print("Enter Plaintext: ", end='')

    cleartext = input()

    print("Enter Cipher Key (16 characters): ", end='')

    cypherkey = input()

    cypherkey = convertToList(cypherkey)

    iv = [103,35,148,239,76,213,47,118,255,222,123,176,106,134,98,92]

    mode, orig\_len, ciph = moo.encrypt(cleartext, moo.modeOfOperation["CBC"],

            cypherkey, moo.aes.keySize["SIZE\_128"], iv)

    print('m=%s, ol=%s (%s), ciph=%s' % (mode, orig\_len, len(cleartext), ciph))

    decr = moo.decrypt(ciph, orig\_len, mode, cypherkey,

            moo.aes.keySize["SIZE\_128"], iv)

    print(decr)

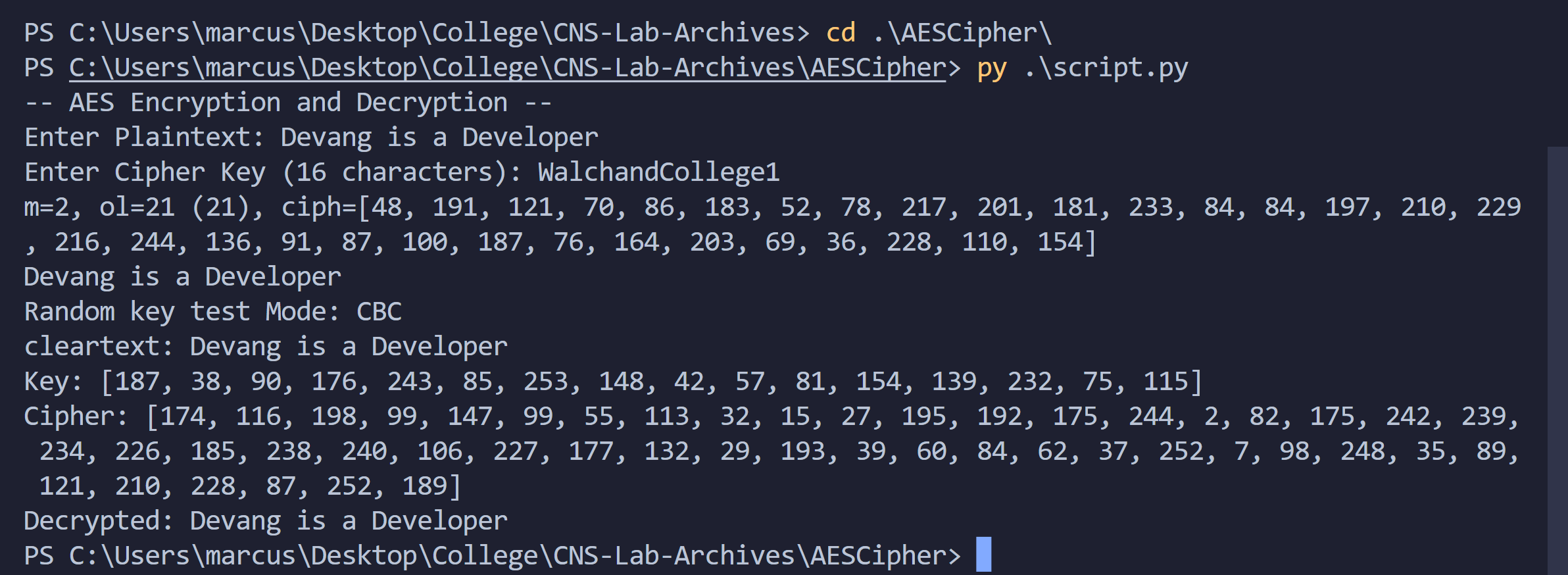
    testStr(cleartext, 16, "CBC")

We now illustrate with examples:

The code is for CBC chaining mode of the AES.

We encrypt – ‘Devang is a Developer’

And let the key be – ‘WalchandCollege1’



As shown, the encryption generated cipher-text as numbers (we don’t convert them to ASCII to avoid vernacular characters).

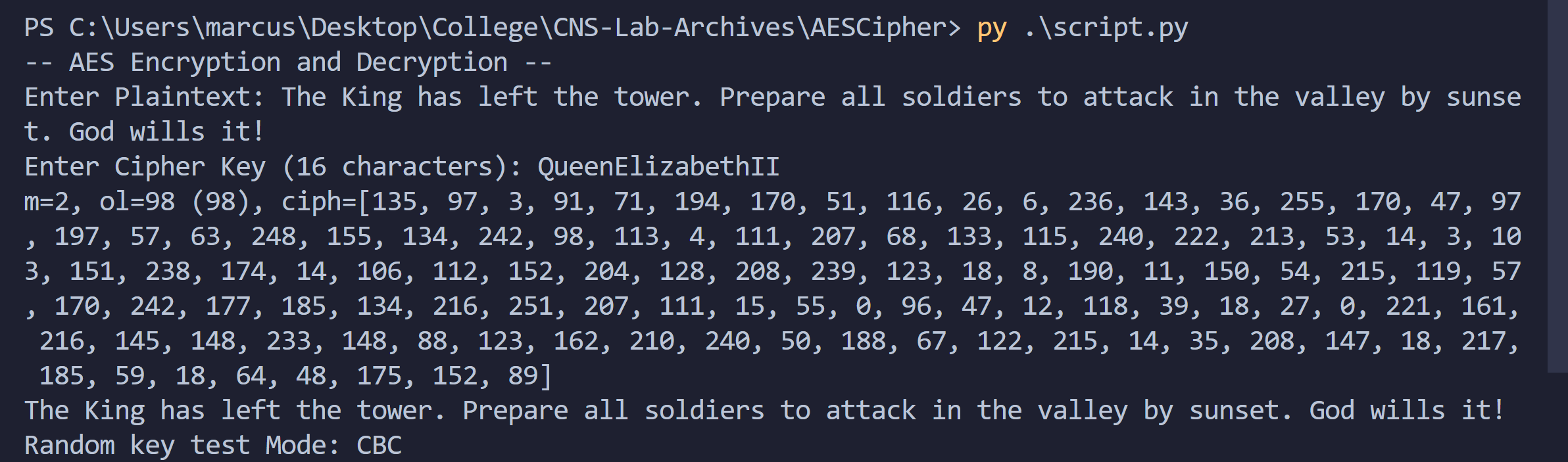
Supplying Key and Ciphertext to the AES decryption algorithm, returns our our plaintext again.

Let’s look at another example, a larger one.

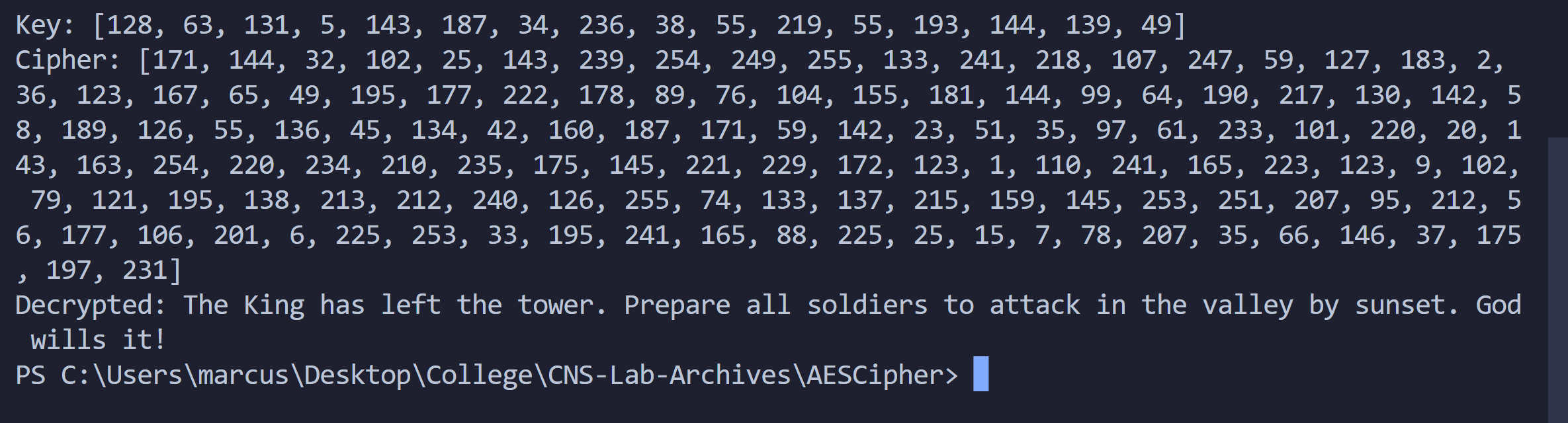
‘The King has left the tower. Prepare all soldiers to attack in the valley by sunset. God wills it!’

Let the key be, ‘QueenElizabethII’

Result of the encryption algorithm:



Decryption Algorithm:



We get our plaintext back!

Thus, we demonstrated the working of the code with examples.

Conclusion:

Thus, the Advanced Encryption Standard (AES) algorithm was studied and demonstrated with the code.