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 :

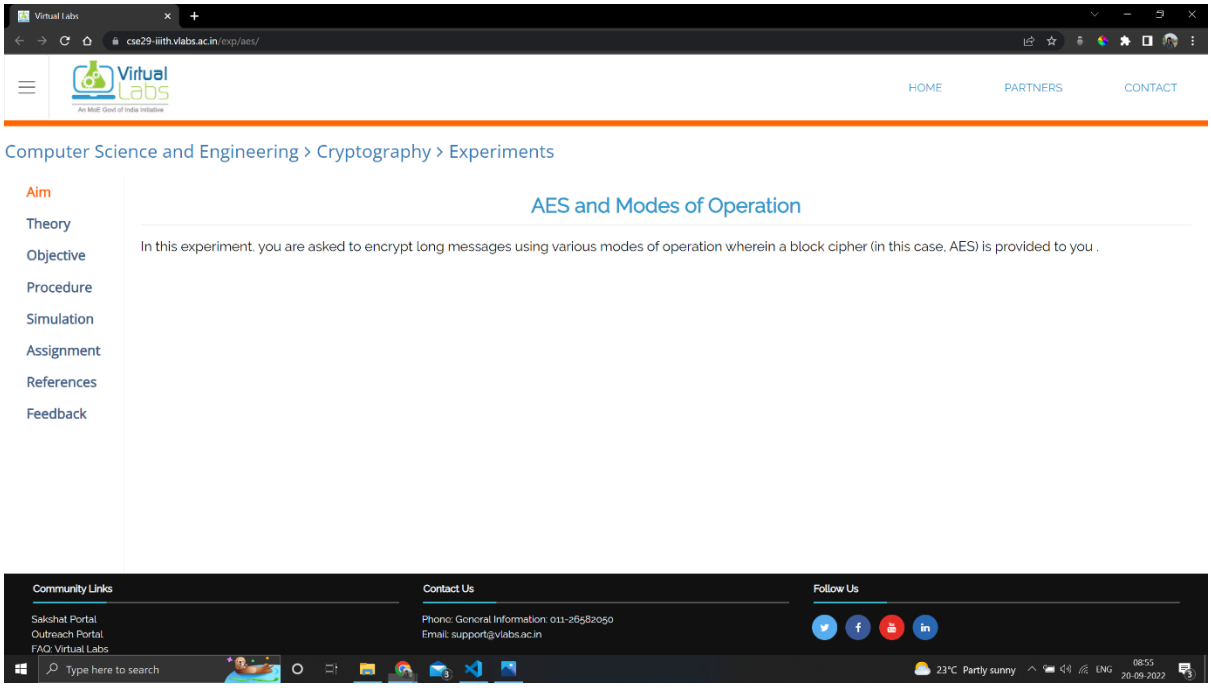
- SubBytes
- ShiftRows
- MixColumns

- Add Round Key

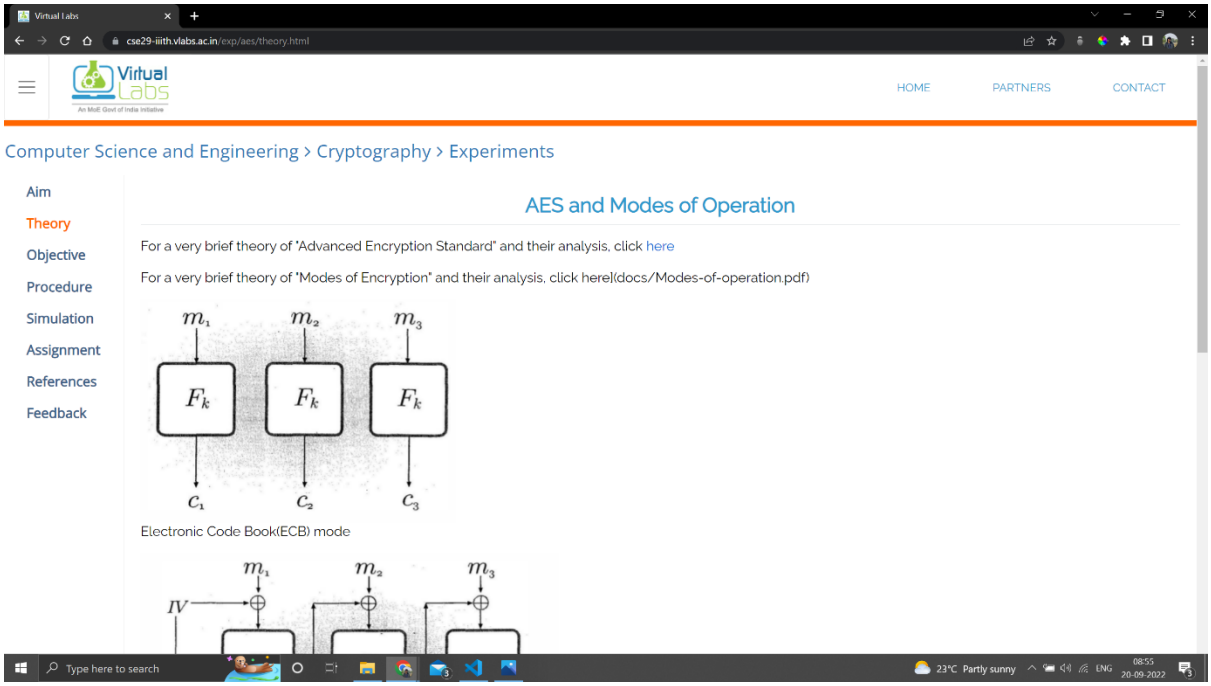
The last round doesn't have the MixColumns round.

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.

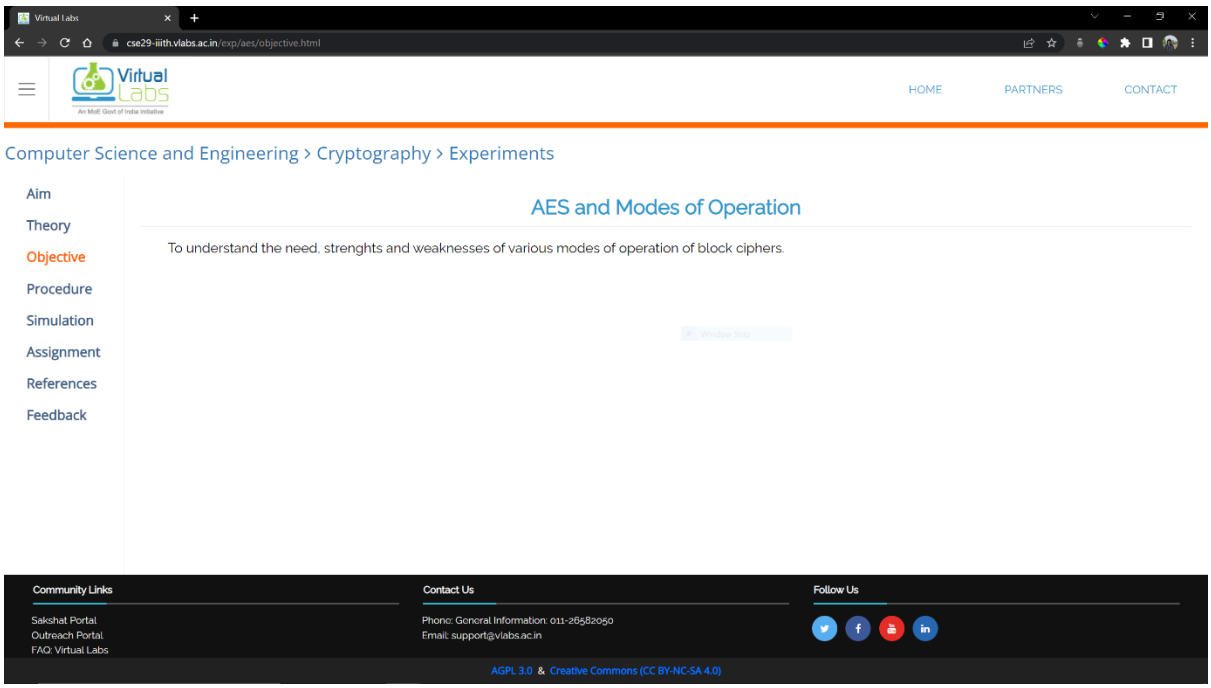
V-Lab Implementation:



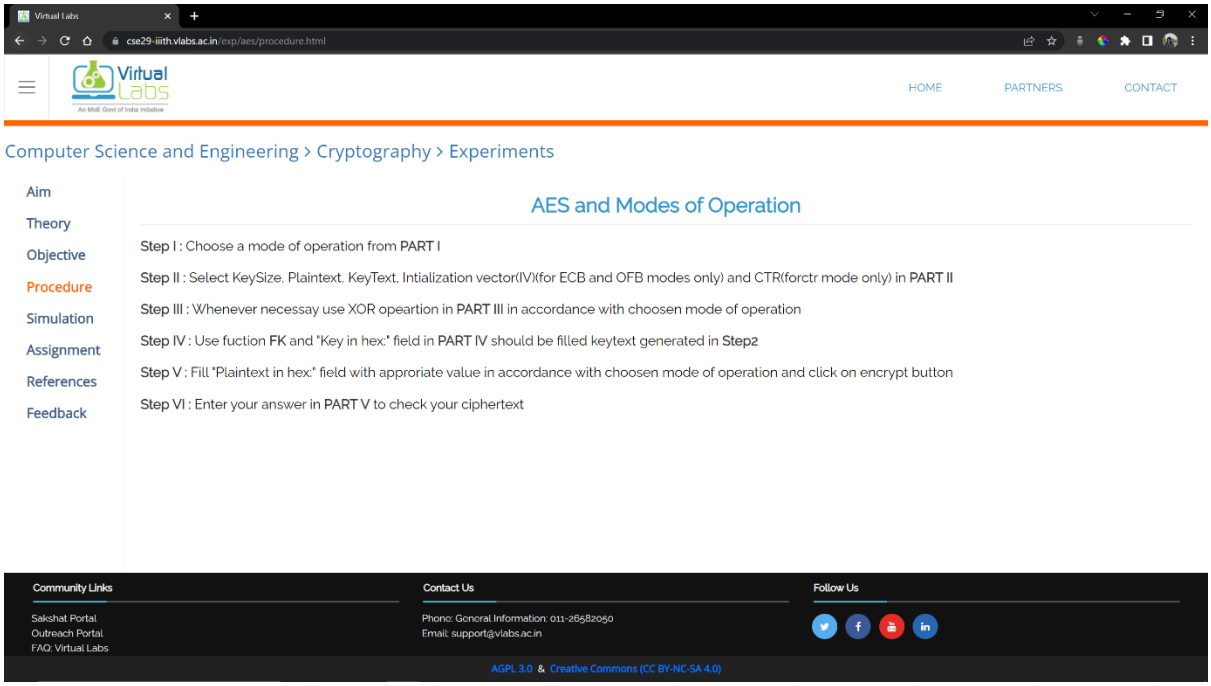
Aim



Theory



Objective

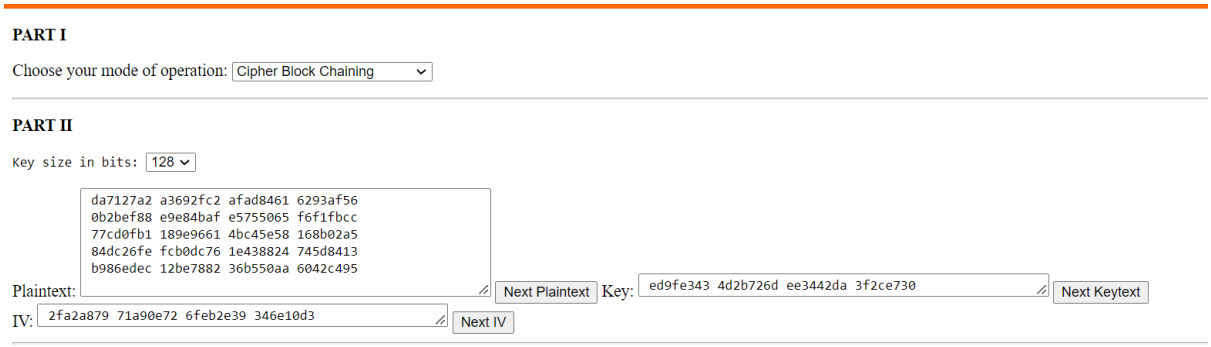


Procedure

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

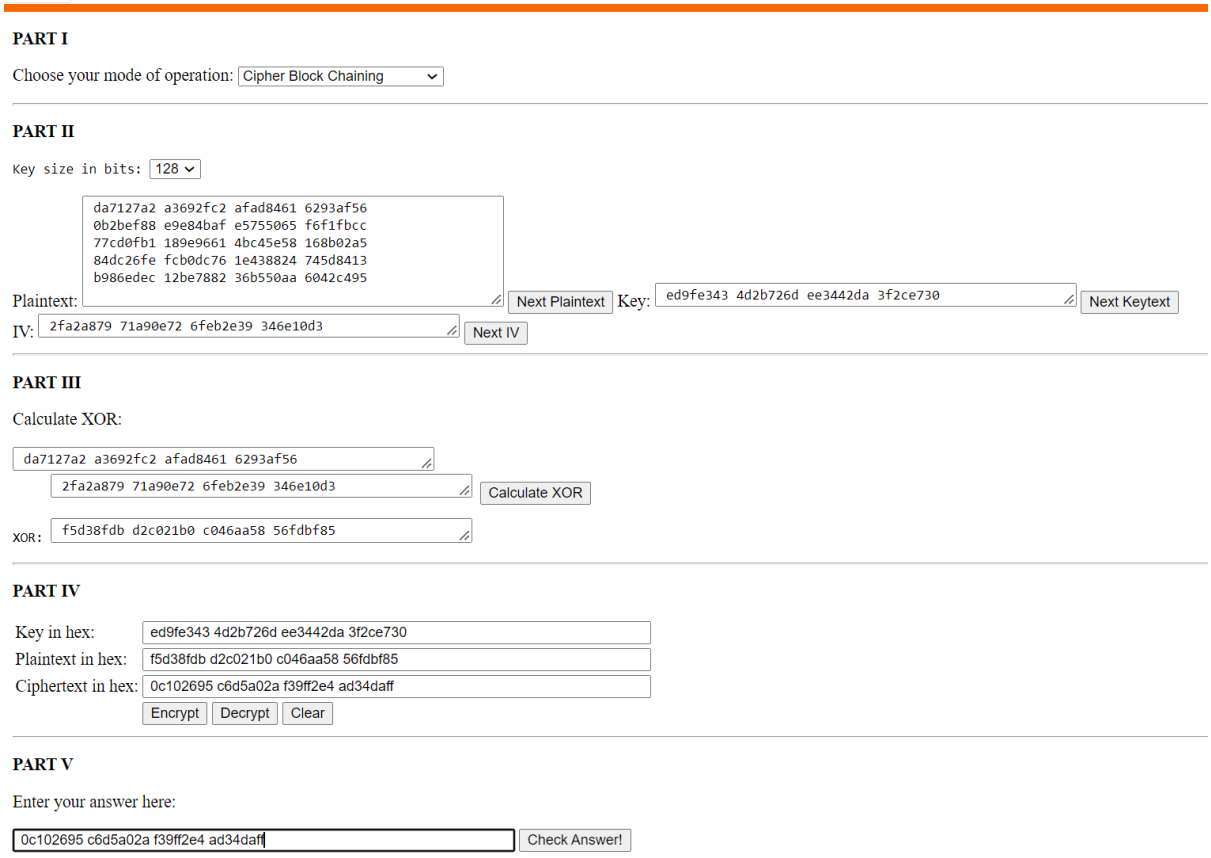
84dc26fe fcb0dc76 1e438824 745d8413

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.

PART I

Choose your mode of operation: Cipher Block Chaining

PART II

Key size in bits: 128

Plaintext:
```
da7127a2 a3692fc2 afad8461 6293af56
0b2bef88 e9e84baf e5755065 f6f1fbcc
77cd0fb1 189e9661 4bc45e58 168b02a5
84dc26fe fcb0dc76 1e438824 745d8413
b986edec 12be7882 36b550aa 6042c495
```
Next Plaintext Key: ed9fe343 4d2b726d ee3442da 3f2ce730 Next Keytext

IV: 2fa2a879 71a90e72 6feb2e39 346e10d3 Next IV

PART III

Calculate XOR:

0b2bef88 e9e84baf e5755065 f6f1fbcc

0c102695 c6d5a02a f39ff2e4 ad34daff Calculate XOR

XOR: 073bc91d 2f3deb85 16eaa281 5bc52133

PART IV

Key in hex: ed9fe343 4d2b726d ee3442da 3f2ce730

Plaintext in hex: 073bc91d 2f3deb85 16eaa281 5bc52133

Ciphertext in hex: b456d318 268261b7 9abd2009 c8240b4f

Encrypt Decrypt Clear

PART V

Enter your answer here:

0c102695 c6d5a02a f39ff2e4 ad34daff b456d318 268261b7 9abd2009 c8240l Check Answer!

PART I

Choose your mode of operation: Cipher Block Chaining

PART II

Key size in bits: 128

Plaintext:
```
da7127a2 a3692fc2 afad8461 6293af56
0b2bef88 e9e84baf e5755065 f6f1fbcc
77cd0fb1 189e9661 4bc45e58 168b02a5
84dc26fe fcb0dc76 1e438824 745d8413
b986edec 12be7882 36b550aa 6042c495
```
Next Plaintext Key: ed9fe343 4d2b726d ee3442da 3f2ce730 Next Keytext

IV: 2fa2a879 71a90e72 6feb2e39 346e10d3 Next IV

PART III

Calculate XOR:

77cd0fb1 189e9661 4bc45e58 168b02a5

b456d318 268261b7 9abd2009 c8240b4f Calculate XOR

XOR: c39bdca9 3e1cf7d6 d1797e51 deaf09ea

PART IV

Key in hex: ed9fe343 4d2b726d ee3442da 3f2ce730

Plaintext in hex: c39bdca9 3e1cf7d6 d1797e51 deaf09ea

Ciphertext in hex: 89801e77 e52dc587 59bd4788 365fa2c1

Encrypt Decrypt Clear

PART V

Enter your answer here:

0c102695 c6d5a02a f39ff2e4 ad34daff b456d318 268261b7 9abd2009 c8240l Check Answer!

PART I

Choose your mode of operation: Cipher Block Chaining

PART II

Key size in bits: 128

Plaintext:
```
da7127a2 a3692fc2 afad8461 6293af56
0b2bef88 e9e84baf e5755065 f6f1fbcc
77cd0fb1 189e9661 4bc45e58 168b02a5
84dc26fe fcb0dc76 1e438824 745d8413
b986edec 12be7882 36b550aa 6042c495
```
Next Plaintext Key: ed9fe343 4d2b726d ee3442da 3f2ce730 Next Keytext

IV: 2fa2a879 71a90e72 6feb2e39 346e10d3 Next IV

PART III

Calculate XOR:

84dc26fe fcb0dc76 1e438824 745d8413

89801e77 e52dc587 59bd4788 365fa2c1 Calculate XOR

XOR: 0d5c3889 199d19f1 47fecfac 420226d2

PART IV

Key in hex: ed9fe343 4d2b726d ee3442da 3f2ce730

Plaintext in hex: 0d5c3889 199d19f1 47fecfac 420226d2

Ciphertext in hex: f02d4d27 38b7552b 374d15e1 a8d4abf8

Encrypt Decrypt Clear

PART V

Enter your answer here:

0c102695 c6d5a02a f39ff2e4 ad34daff b456d318 268261b7 9abd2009 c8240| Check Answer!

PART I

Choose your mode of operation: Cipher Block Chaining

PART II

Key size in bits: 128

Plaintext:
```
da7127a2 a3692fc2 afad8461 6293af56
0b2bef88 e9e84baf e5755065 f6f1fbcc
77cd0fb1 189e9661 4bc45e58 168b02a5
84dc26fe fcb0dc76 1e438824 745d8413
b986edec 12be7882 36b550aa 6042c495
```
Next Plaintext Key: ed9fe343 4d2b726d ee3442da 3f2ce730 Next Keytext

IV: 2fa2a879 71a90e72 6feb2e39 346e10d3 Next IV

PART III

Calculate XOR:

b986edec 12be7882 36b550aa 6042c495

f02d4d27 38b7552b 374d15e1 a8d4abf8 Calculate XOR

XOR: 49aba0cb 2a092da9 01f8454b c8966f6d

PART IV

Key in hex: ed9fe343 4d2b726d ee3442da 3f2ce730

Plaintext in hex: 49aba0cb 2a092da9 01f8454b c8966f6d

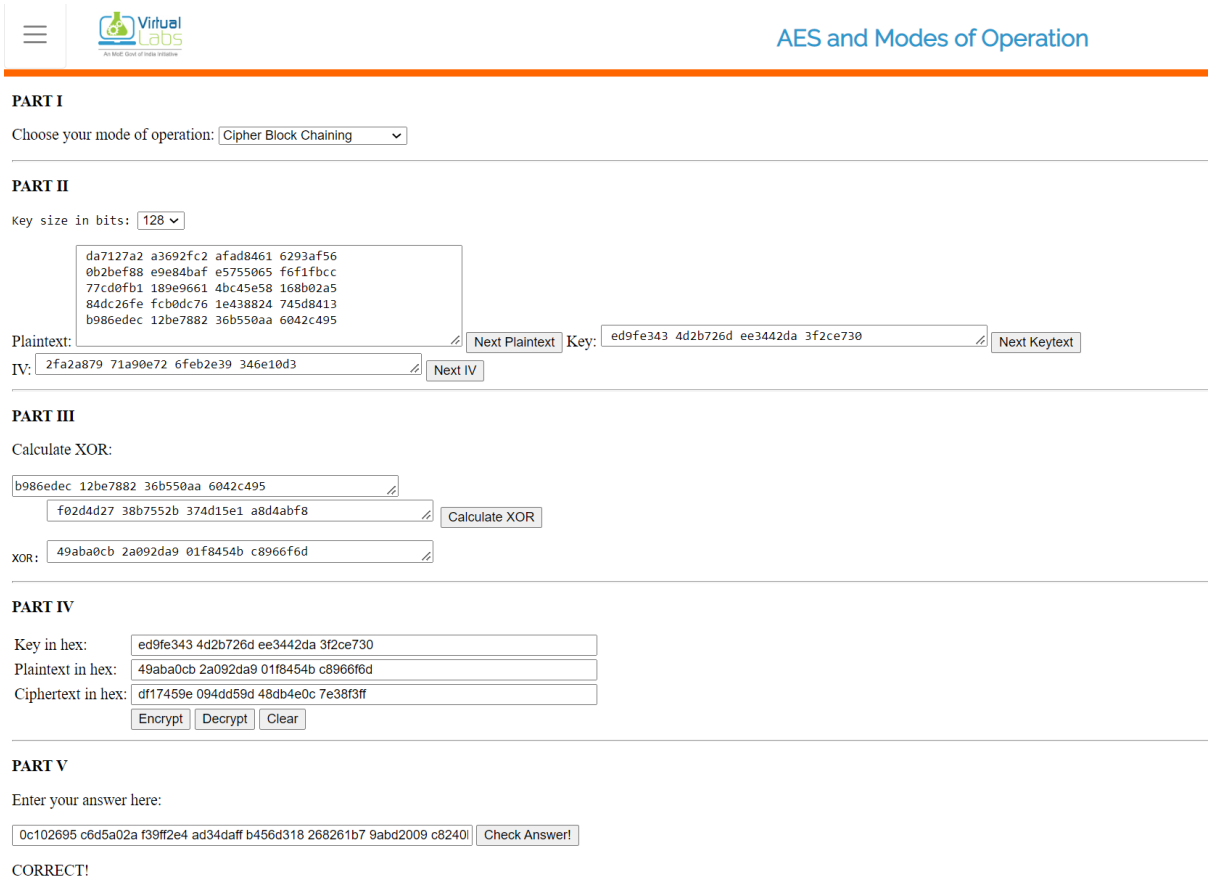
Ciphertext in hex: df17459e 094dd59d 48db4e0c 7e38f3ff

Encrypt Decrypt Clear

PART V

Enter your answer here:

0c102695 c6d5a02a f39ff2e4 ad34daff b456d318 268261b7 9abd2009 c8240| Check Answer!



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 =  [0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67,
            0x2b, 0xfe, 0xd7, 0xab, 0x76, 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59,
            0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, 0xb7,
            0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1,
            0x71, 0xd8, 0x31, 0x15, 0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05,
            0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, 0x09, 0x83,
            0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29,
            0xe3, 0x2f, 0x84, 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
            0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, 0xd0, 0xef, 0xaa,
            0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c,
            0x9f, 0xa8, 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc,
            0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, 0xcd, 0x0c, 0x13, 0xec,
            0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19,
            0x73, 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee,
            0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, 0xe0, 0x32, 0x3a, 0x0a, 0x49,
            0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
            0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4,
            0xea, 0x65, 0x7a, 0xae, 0x08, 0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6,
            0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, 0x70,
            0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9,
            0x86, 0xc1, 0x1d, 0x9e, 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e,
            0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, 0x8c, 0xa1,
            0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0,
            0x54, 0xbb, 0x16]

    # Rijndael Inverted S-box
    rsbox = [0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3,
            0x9e, 0x81, 0xf3, 0xd7, 0xfb , 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f,
            0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb , 0x54,
            0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b,
            0x42, 0xfa, 0xc3, 0x4e , 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24,
            0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25 , 0x72, 0xf8,
            0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d,
            0x65, 0xb6, 0x92 , 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda,
            0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84 , 0x90, 0xd8, 0xab,
            0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3,
            0x45, 0x06 , 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1,
            0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b , 0x3a, 0x91, 0x11, 0x41,
            0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6,
            0x73 , 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9,
            0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e , 0x47, 0xf1, 0x1a, 0x71, 0x1d,
```

```
            0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b ,
            0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0,
            0xfe, 0x78, 0xcd, 0x5a, 0xf4 , 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07,
            0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f , 0x60,
            0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f,
            0x93, 0xc9, 0x9c, 0xef , 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5,
            0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61 , 0x17, 0x2b,
            0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55,
            0x21, 0x0c, 0x7d]

    def 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 = [0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36,
            0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97,
            0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72,
            0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66,
            0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
            0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,
            0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
            0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61,
            0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,
            0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40,
            0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc,
            0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5,
            0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a,
            0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d,
            0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c,
            0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,
            0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4,
            0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,
            0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08,
            0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
            0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
```

```
            0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2,
            0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74,
            0xe8, 0xcb ]

    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 & 0x80
            a <<= 1
            # keep a 8 bit
            a &= 0xFF
            if hi_bit_set:
                a ^= 0x1b
            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’

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives> cd .\AESCipher\
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\AESCipher> py .\script.py
-- AES Encryption and Decryption --
Enter Plaintext: Devang is a Developer
Enter Cipher Key (16 characters): WalchandCollege1
m=2, ol=21 (21), ciph=[48, 191, 121, 70, 86, 183, 52, 78, 217, 201, 181, 233, 84, 84, 197, 210, 229
, 216, 244, 136, 91, 87, 100, 187, 76, 164, 203, 69, 36, 228, 110, 154]
Devang is a Developer
Random key test Mode: CBC
cleartext: Devang is a Developer
Key: [187, 38, 90, 176, 243, 85, 253, 148, 42, 57, 81, 154, 139, 232, 75, 115]
Cipher: [174, 116, 198, 99, 147, 99, 55, 113, 32, 15, 27, 195, 192, 175, 244, 2, 82, 175, 242, 239,
 234, 226, 185, 238, 240, 106, 227, 177, 132, 29, 193, 39, 60, 84, 62, 37, 252, 7, 98, 248, 35, 89,
 121, 210, 228, 87, 252, 189]
Decrypted: Devang is a Developer
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\AESCipher>
```

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:

```
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\AESCipher> py .\script.py
-- AES Encryption and Decryption --
Enter Plaintext: The King has left the tower. Prepare all soldiers to attack in the valley by sunse
t. God wills it!
Enter Cipher Key (16 characters): QueenElizabethII
m=2, ol=98 (98), ciph=[135, 97, 3, 91, 71, 194, 170, 51, 116, 26, 6, 236, 143, 36, 255, 170, 47, 97
, 197, 57, 63, 248, 155, 134, 242, 98, 113, 4, 111, 207, 68, 133, 115, 240, 222, 213, 53, 14, 3, 10
3, 151, 238, 174, 14, 106, 112, 152, 204, 128, 208, 239, 123, 18, 8, 190, 11, 150, 54, 215, 119, 57
, 170, 242, 177, 185, 134, 216, 251, 207, 111, 15, 55, 0, 96, 47, 12, 118, 39, 18, 27, 0, 221, 161,
 216, 145, 148, 233, 148, 88, 123, 162, 210, 240, 50, 188, 67, 122, 215, 14, 35, 208, 147, 18, 217,
 185, 59, 18, 64, 48, 175, 152, 89]
The King has left the tower. Prepare all soldiers to attack in the valley by sunset. God wills it!
Random key test Mode: CBC
```

Decryption Algorithm:

```
Key: [128, 63, 131, 5, 143, 187, 34, 236, 38, 55, 219, 55, 193, 144, 139, 49]
Cipher: [171, 144, 32, 102, 25, 143, 239, 254, 249, 255, 133, 241, 218, 107, 247, 59, 127, 183, 2,
36, 123, 167, 65, 49, 195, 177, 222, 178, 89, 76, 104, 155, 181, 144, 99, 64, 190, 217, 130, 142, 5
8, 189, 126, 55, 136, 45, 134, 42, 160, 187, 171, 59, 142, 23, 51, 35, 97, 61, 233, 101, 220, 20, 1
43, 163, 254, 220, 234, 210, 235, 175, 145, 221, 229, 172, 123, 1, 110, 241, 165, 223, 123, 9, 102,
 79, 121, 195, 138, 213, 212, 240, 126, 255, 74, 133, 137, 215, 159, 145, 253, 251, 207, 95, 212, 5
6, 177, 106, 201, 6, 225, 253, 33, 195, 241, 165, 88, 225, 25, 15, 7, 78, 207, 35, 66, 146, 37, 175
, 197, 231]
Decrypted: The King has left the tower. Prepare all soldiers to attack in the valley by sunset. God
 wills it!
PS C:\Users\marcus\Desktop\College\CNS-Lab-Archives\AESCipher>
```

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.