CODE

SERVER

import socket

from aes\_implementation import AES

def start\_server():

HOST = '127.0.0.1'

PORT = 65432

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

s.bind((HOST, PORT))

s.listen()

print(f"Server listening on {HOST}:{PORT}")

while True:

conn, addr = s.accept()

with conn:

print(f"Connected by {addr}")

\

key\_data = conn.recv(16)

key = list(key\_data)

data = conn.recv(16)

plaintext = list(data)

aes = AES(key)

ciphertext = aes.encrypt(plaintext)

print("Received Plaintext:", plaintext)

print("Encrypted Ciphertext:", ciphertext)

conn.sendall(bytes(ciphertext))

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

start\_server()

CLIENT

import socket

from aes\_implementation import AES

def start\_client():

HOST = '127.0.0.1'

PORT = 65432

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

s.connect((HOST, PORT))

key = bytes([

0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,

0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c

])

s.sendall(key)

plaintext = bytes([

0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,

0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a

])

s.sendall(plaintext)

ciphertext = s.recv(16)

print("Sent Plaintext:", list(plaintext))

print("Received Ciphertext:", list(ciphertext))

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

start\_client()

AES\_IMPLEMENTATION

import numpy as np

class AES:

def \_\_init\_\_(self, key):

self.Sbox = self.generate\_sbox()

self.InvSbox = self.generate\_inv\_sbox()

self.Rcon = [0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36]

self.key = self.key\_expansion(key)

def generate\_sbox(self):

# Standard AES Sbox

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

]

return sbox

def generate\_inv\_sbox(self):

inv\_sbox = np.zeros(256, dtype=np.uint8)

for i in range(256):

inv\_sbox[self.Sbox[i]] = i

return inv\_sbox

def multiplicative\_inverse(self, b):

if b == 0:

return 0

a = 0x11b

for \_ in range(8):

if b & 1:

a ^= b

b >>= 1

if b == 0:

break

return a

def affine\_transform(self, x):

c = 0b01100011

transformed = 0

for i in range(8):

transformed |= (bin(x).count('1') % 2 ^ ((c >> i) & 1)) << i

return transformed

def key\_expansion(self, key):

key\_length = len(key)

Nk = key\_length // 4

Nr = Nk + 6

key\_schedule = [0] \* (4 \* (Nr + 1) \* 4)

for i in range(key\_length):

key\_schedule[i] = key[i]

for i in range(Nk, 4 \* (Nr + 1)):

temp = key\_schedule[(i-1)\*4 : i\*4]

if i % Nk == 0:

temp = temp[1:] + temp[:1]

temp = [self.Sbox[x] for x in temp]

temp[0] ^= self.Rcon[i // Nk]

for j in range(4):

key\_schedule[i\*4 + j] = key\_schedule[(i-Nk)\*4 + j] ^ temp[j]

return key\_schedule

def sub\_bytes(self, state):

return [self.Sbox[x] for x in state]

def inv\_sub\_bytes(self, state):

return [self.InvSbox[x] for x in state]

def shift\_rows(self, state):

state\_matrix = [state[i:i+4] for i in range(0, 16, 4)]

state\_matrix[1] = state\_matrix[1][1:] + state\_matrix[1][:1]

state\_matrix[2] = state\_matrix[2][2:] + state\_matrix[2][:2]

state\_matrix[3] = state\_matrix[3][3:] + state\_matrix[3][:3]

return [x for row in state\_matrix for x in row]

def inv\_shift\_rows(self, state):

state\_matrix = [state[i:i+4] for i in range(0, 16, 4)]

state\_matrix[1] = state\_matrix[1][-1:] + state\_matrix[1][:-1]

state\_matrix[2] = state\_matrix[2][-2:] + state\_matrix[2][:-2]

state\_matrix[3] = state\_matrix[3][-3:] + state\_matrix[3][:-3]

return [x for row in state\_matrix for x in row]

def mix\_columns(self, state):

mixed\_state = [0] \* 16

for col in range(4):

column = state[col\*4 : col\*4+4]

mixed\_state[col\*4] = (self.gmul(column[0], 2) ^

self.gmul(column[3], 1) ^

self.gmul(column[2], 1) ^

self.gmul(column[1], 3))

mixed\_state[col\*4+1] = (self.gmul(column[1], 2) ^

self.gmul(column[0], 1) ^

self.gmul(column[3], 1) ^

self.gmul(column[2], 3))

mixed\_state[col\*4+2] = (self.gmul(column[2], 2) ^

self.gmul(column[1], 1) ^

self.gmul(column[0], 1) ^

self.gmul(column[3], 3))

mixed\_state[col\*4+3] = (self.gmul(column[3], 2) ^

self.gmul(column[2], 1) ^

self.gmul(column[1], 1) ^

self.gmul(column[0], 3))

return mixed\_state

def gmul(self, a, b):

p = 0

for \_ in range(8):

if b & 1:

p ^= a

hi\_bit\_set = a & 0x80

a <<= 1

if hi\_bit\_set:

a ^= 0x1B

a &= 0xFF

b >>= 1

return p

def add\_round\_key(self, state, round\_key):

return [state[i] ^ round\_key[i] for i in range(16)]

def encrypt(self, plaintext):

while len(plaintext) < 16:

plaintext.append(0)

state = plaintext.copy()

Nr = len(self.key) // 16 - 1

state = self.add\_round\_key(state, self.key[:16])

for round in range(1, Nr + 1):

state = self.sub\_bytes(state)

state = self.shift\_rows(state)

state = self.mix\_columns(state)

state = self.add\_round\_key(state, self.key[round\*16:(round+1)\*16])

state = self.sub\_bytes(state)

state = self.shift\_rows(state)

state = self.add\_round\_key(state, self.key[-16:])

return state

def decrypt(self, ciphertext):

state = ciphertext.copy()

Nr = len(self.key) // 16 - 1

state = self.add\_round\_key(state, self.key[-16:])

for round in range(Nr, 0, -1):

state = self.inv\_shift\_rows(state)

state = self.inv\_sub\_bytes(state)

state = self.add\_round\_key(state, self.key[round\*16:(round+1)\*16])

state = self.mix\_columns(state)

state = self.inv\_shift\_rows(state)

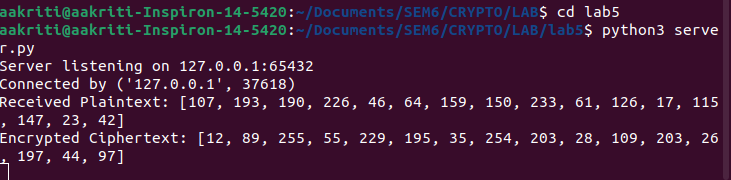
state = self.inv\_sub\_bytes(state)

state = self.add\_round\_key(state, self.key[:16])

return state

OUTPUT

SERVER



CLIENT

