#!/usr/bin/env python  
## Homework Number: HW4  
## Name: ZhiFei Chen  
## ECN Login: chen2281  
## Due Date: 2/15/2020  
  
  
import sys  
import os  
import BitVector  
from BitVector import \*  
  
AES\_modulus = BitVector(bitstring='100011011')  
  
  
subBytesTable = [] # for encryption  
invSubBytesTable = [] # for decryption  
  
def genTables():  
 c = BitVector(bitstring='01100011')  
 d = BitVector(bitstring='00000101')  
 for i in range(0, 256):  
 # For the encryption SBox  
 a = BitVector(intVal = i, size=8).gf\_MI(AES\_modulus, 8) if i != 0 else BitVector(intVal=0)  
 # For bit scrambling for the encryption SBox entries:  
 a1,a2,a3,a4 = [a.deep\_copy() for x in range(4)]  
 a ^= (a1 >> 4) ^ (a2 >> 5) ^ (a3 >> 6) ^ (a4 >> 7) ^ c  
 subBytesTable.append(int(a))  
 # For the decryption Sbox:  
 b = BitVector(intVal = i, size=8)  
 # For bit scrambling for the decryption SBox entries:  
 b1,b2,b3 = [b.deep\_copy() for x in range(3)]  
 b = (b1 >> 2) ^ (b2 >> 5) ^ (b3 >> 7) ^ d  
 check = b.gf\_MI(AES\_modulus, 8)  
 b = check if isinstance(check, BitVector) else 0  
 invSubBytesTable.append(int(b))  
  
def gee(keyword, round\_constant, byte\_sub\_table):  
 *'''  
 This is the g() function you see in Figure 4 of Lecture 8.  
 '''* rotated\_word = keyword.deep\_copy()  
 rotated\_word << 8  
 newword = BitVector(size = 0)  
 for i in range(4):  
 newword += BitVector(intVal = byte\_sub\_table[rotated\_word[8\*i:8\*i+8].intValue()], size = 8)  
 newword[:8] ^= round\_constant  
 round\_constant = round\_constant.gf\_multiply\_modular(BitVector(intVal = 0x02), AES\_modulus, 8)  
 return newword, round\_constant  
  
def gen\_key\_schedule\_128(key\_bv):  
 byte\_sub\_table = gen\_subbytes\_table()  
 # We need 44 keywords in the key schedule for 128 bit AES. Each keyword is 32-bits  
 # wide. The 128-bit AES uses the first four keywords to xor the input block with.  
 # Subsequently, each of the 10 rounds uses 4 keywords from the key schedule. We will  
 # store all 44 keywords in the following list:  
 key\_words = [None for i in range(44)]  
 round\_constant = BitVector(intVal = 0x01, size=8)  
 for i in range(4):  
 key\_words[i] = key\_bv[i\*32 : i\*32 + 32]  
 for i in range(4,44):  
 if i%4 == 0:  
 kwd, round\_constant = gee(key\_words[i-1], round\_constant, byte\_sub\_table)  
 key\_words[i] = key\_words[i-4] ^ kwd  
 else:  
 key\_words[i] = key\_words[i-4] ^ key\_words[i-1]  
 return key\_words  
  
def gen\_key\_schedule\_192(key\_bv):  
 byte\_sub\_table = gen\_subbytes\_table()  
 # We need 52 keywords (each keyword consists of 32 bits) in the key schedule for  
 # 192 bit AES. The 192-bit AES uses the first four keywords to xor the input  
 # block with. Subsequently, each of the 12 rounds uses 4 keywords from the key  
 # schedule. We will store all 52 keywords in the following list:  
 key\_words = [None for i in range(52)]  
 round\_constant = BitVector(intVal = 0x01, size=8)  
 for i in range(6):  
 key\_words[i] = key\_bv[i\*32 : i\*32 + 32]  
 for i in range(6,52):  
 if i%6 == 0:  
 kwd, round\_constant = gee(key\_words[i-1], round\_constant, byte\_sub\_table)  
 key\_words[i] = key\_words[i-6] ^ kwd  
 else:  
 key\_words[i] = key\_words[i-6] ^ key\_words[i-1]  
 return key\_words  
  
def gen\_key\_schedule\_256(key\_bv):  
 byte\_sub\_table = gen\_subbytes\_table()  
 # We need 60 keywords (each keyword consists of 32 bits) in the key schedule for  
 # 256 bit AES. The 256-bit AES uses the first four keywords to xor the input  
 # block with. Subsequently, each of the 14 rounds uses 4 keywords from the key  
 # schedule. We will store all 60 keywords in the following list:  
 key\_words = [None for i in range(60)]  
 round\_constant = BitVector(intVal = 0x01, size=8)  
 for i in range(8):  
 key\_words[i] = key\_bv[i\*32 : i\*32 + 32]  
 for i in range(8,60):  
 if i%8 == 0:  
 kwd, round\_constant = gee(key\_words[i-1], round\_constant, byte\_sub\_table)  
 key\_words[i] = key\_words[i-8] ^ kwd  
 elif (i - (i//8)\*8) < 4:  
 key\_words[i] = key\_words[i-8] ^ key\_words[i-1]  
 elif (i - (i//8)\*8) == 4:  
 key\_words[i] = BitVector(size = 0)  
 for j in range(4):  
 key\_words[i] += BitVector(intVal =  
 byte\_sub\_table[key\_words[i-1][8\*j:8\*j+8].intValue()], size = 8)  
 key\_words[i] ^= key\_words[i-8]  
 elif ((i - (i//8)\*8) > 4) and ((i - (i//8)\*8) < 8):  
 key\_words[i] = key\_words[i-8] ^ key\_words[i-1]  
 else:  
 sys.exit("error in key scheduling algo for i = %d" % i)  
 return key\_words  
  
def gen\_subbytes\_table():  
 subBytesTable = []  
 c = BitVector(bitstring='01100011')  
 for i in range(0, 256):  
 a = BitVector(intVal = i, size=8).gf\_MI(AES\_modulus, 8) if i != 0 else BitVector(intVal=0)  
 a1,a2,a3,a4 = [a.deep\_copy() for x in range(4)]  
 a ^= (a1 >> 4) ^ (a2 >> 5) ^ (a3 >> 6) ^ (a4 >> 7) ^ c  
 subBytesTable.append(int(a))  
 return subBytesTable  
  
def gen\_key():  
 key\_words = []  
 keysize = 256  
 key = open(sys.argv[3], 'r').read()  
 key\_bv = BitVector(textstring= key) # pass the key to the bitvector class  
 if keysize == 128:  
 key\_words = gen\_key\_schedule\_128(key\_bv)  
 elif keysize == 192:  
 key\_words = gen\_key\_schedule\_192(key\_bv)  
 elif keysize == 256:  
 key\_words = gen\_key\_schedule\_256(key\_bv)  
 else:  
 sys.exit("wrong keysize --- aborting")  
  
 key\_schedule = []  
 for word\_index,word in enumerate(key\_words):  
 keyword\_in\_ints = []  
 for i in range(4):  
 keyword\_in\_ints.append(word[i\*8:i\*8+8].intValue())  
 key\_schedule.append(keyword\_in\_ints)  
  
 num\_rounds = None  
 if keysize == 128: num\_rounds = 10  
 if keysize == 192: num\_rounds = 12  
 if keysize == 256: num\_rounds = 14  
 round\_keys = [None for i in range(num\_rounds+1)]  
 for i in range(num\_rounds+1):  
 round\_keys[i] = (key\_words[i\*4] + key\_words[i\*4+1] + key\_words[i\*4+2] + key\_words[i\*4+3]).get\_bitvector\_in\_hex()  
  
 return key\_schedule, round\_keys  
  
############  
def sub\_bytes(state\_array, s\_table):  
 for i in range(4):  
 for j in range(4):  
 state\_array[i][j] = BitVector(intVal=s\_table[int(state\_array[i][j])], size=8)  
  
 #print(state\_array[i][j])  
  
 return state\_array  
  
def shift\_rows(state\_array):  
 temp = state\_array[0][0 : 4]  
 #print(temp[3])  
 state\_array[0][0 : 4] = temp[0:]+temp[0:0]  
  
 temp1 = state\_array[1][0 : 4]  
 state\_array[1][0 : 4] = temp1[1:]+temp1[0:1]  
  
 temp2 = state\_array[2][0 : 4]  
 state\_array[2][0 : 4] = temp2[2:]+temp2[0:2]  
  
 temp3 = state\_array[3][0 : 4]  
 state\_array[3][0:4] = temp3[3:] + temp3[0:3]  
  
 return state\_array  
  
def inv\_shift\_rows(state\_array):  
 temp = state\_array[0][0: 4]  
 # print(temp[3])  
 state\_array[0][0: 4] = temp[0:] + temp[0:0]  
  
 temp1 = state\_array[1][0: 4]  
 state\_array[1][0: 4] = temp1[3:] + temp1[0:3]  
  
 temp2 = state\_array[2][0: 4]  
 state\_array[2][0: 4] = temp2[2:] + temp2[0:2]  
  
 temp3 = state\_array[3][0: 4]  
 state\_array[3][0:4] = temp3[1:] + temp3[0:1]  
  
 return state\_array  
  
def mix\_column(state\_array):  
 multiply\_maxtrix = [[BitVector(intVal=0x00, size=8) for x in range(4)] for x in range(4)]  
 result = [[0 for x in range(4)] for x in range(4)]  
 multiply\_maxtrix[0][0] = BitVector(intVal=0x02, size=8)  
 multiply\_maxtrix[0][1] = BitVector(intVal=0x03, size=8)  
 multiply\_maxtrix[0][2] = BitVector(intVal=0x01, size=8)  
 multiply\_maxtrix[0][3] = BitVector(intVal=0x01, size=8)  
  
 multiply\_maxtrix[1][0] = BitVector(intVal=0x01, size=8)  
 multiply\_maxtrix[1][1] = BitVector(intVal=0x02, size=8)  
 multiply\_maxtrix[1][2] = BitVector(intVal=0x03, size=8)  
 multiply\_maxtrix[1][3] = BitVector(intVal=0x01, size=8)  
  
 multiply\_maxtrix[2][0] = BitVector(intVal=0x01, size=8)  
 multiply\_maxtrix[2][1] = BitVector(intVal=0x01, size=8)  
 multiply\_maxtrix[2][2] = BitVector(intVal=0x02, size=8)  
 multiply\_maxtrix[2][3] = BitVector(intVal=0x03, size=8)  
  
 multiply\_maxtrix[3][0] = BitVector(intVal=0x03, size=8)  
 multiply\_maxtrix[3][1] = BitVector(intVal=0x01, size=8)  
 multiply\_maxtrix[3][2] = BitVector(intVal=0x01, size=8)  
 multiply\_maxtrix[3][3] = BitVector(intVal=0x02, size=8)  
  
  
  
 for j in range(4):  
 result[0][j] = state\_array[0][j].gf\_multiply\_modular(multiply\_maxtrix[0][0], AES\_modulus, 8) ^ \  
 state\_array[1][j].gf\_multiply\_modular(multiply\_maxtrix[0][1], AES\_modulus, 8) ^ \  
 state\_array[2][j] ^ \  
 state\_array[3][j]  
  
 result[1][j] = state\_array[0][j].gf\_multiply\_modular(multiply\_maxtrix[1][0], AES\_modulus, 8) ^ \  
 state\_array[1][j].gf\_multiply\_modular(multiply\_maxtrix[1][1], AES\_modulus, 8) ^ \  
 state\_array[2][j].gf\_multiply\_modular(multiply\_maxtrix[1][2], AES\_modulus, 8) ^ \  
 state\_array[3][j]  
  
 result[2][j] = state\_array[0][j].gf\_multiply\_modular(multiply\_maxtrix[2][0], AES\_modulus, 8) ^ \  
 state\_array[1][j].gf\_multiply\_modular(multiply\_maxtrix[2][1], AES\_modulus, 8) ^ \  
 state\_array[2][j].gf\_multiply\_modular(multiply\_maxtrix[2][2], AES\_modulus, 8) ^ \  
 state\_array[3][j].gf\_multiply\_modular(multiply\_maxtrix[2][3], AES\_modulus, 8)  
  
 result[3][j] = state\_array[0][j].gf\_multiply\_modular(multiply\_maxtrix[3][0], AES\_modulus, 8) ^ \  
 state\_array[1][j].gf\_multiply\_modular(multiply\_maxtrix[3][1], AES\_modulus, 8) ^ \  
 state\_array[2][j].gf\_multiply\_modular(multiply\_maxtrix[3][2], AES\_modulus, 8) ^ \  
 state\_array[3][j].gf\_multiply\_modular(multiply\_maxtrix[3][3], AES\_modulus, 8)  
  
  
 return result  
  
def inv\_mix\_column(state\_array):  
 multiply\_maxtrix = [[BitVector(intVal=0x00, size=8) for x in range(4)] for x in range(4)]  
 result = [[0 for x in range(4)] for x in range(4)]  
 multiply\_maxtrix[0][0] = BitVector(intVal=0x0E, size=8)  
 multiply\_maxtrix[0][1] = BitVector(intVal=0x0B, size=8)  
 multiply\_maxtrix[0][2] = BitVector(intVal=0x0D, size=8)  
 multiply\_maxtrix[0][3] = BitVector(intVal=0x09, size=8)  
  
 multiply\_maxtrix[1][0] = BitVector(intVal=0x09, size=8)  
 multiply\_maxtrix[1][1] = BitVector(intVal=0x0E, size=8)  
 multiply\_maxtrix[1][2] = BitVector(intVal=0x0B, size=8)  
 multiply\_maxtrix[1][3] = BitVector(intVal=0x0D, size=8)  
  
 multiply\_maxtrix[2][0] = BitVector(intVal=0x0D, size=8)  
 multiply\_maxtrix[2][1] = BitVector(intVal=0x09, size=8)  
 multiply\_maxtrix[2][2] = BitVector(intVal=0x0E, size=8)  
 multiply\_maxtrix[2][3] = BitVector(intVal=0x0B, size=8)  
  
 multiply\_maxtrix[3][0] = BitVector(intVal=0x0B, size=8)  
 multiply\_maxtrix[3][1] = BitVector(intVal=0x0D, size=8)  
 multiply\_maxtrix[3][2] = BitVector(intVal=0x09, size=8)  
 multiply\_maxtrix[3][3] = BitVector(intVal=0x0E, size=8)  
  
 for j in range(4):  
 result[0][j] = state\_array[0][j].gf\_multiply\_modular(multiply\_maxtrix[0][0], AES\_modulus, 8) ^ \  
 state\_array[1][j].gf\_multiply\_modular(multiply\_maxtrix[0][1], AES\_modulus, 8) ^ \  
 state\_array[2][j].gf\_multiply\_modular(multiply\_maxtrix[0][2], AES\_modulus, 8) ^ \  
 state\_array[3][j].gf\_multiply\_modular(multiply\_maxtrix[0][3], AES\_modulus, 8)  
  
 result[1][j] = state\_array[0][j].gf\_multiply\_modular(multiply\_maxtrix[1][0], AES\_modulus, 8) ^ \  
 state\_array[1][j].gf\_multiply\_modular(multiply\_maxtrix[1][1], AES\_modulus, 8) ^ \  
 state\_array[2][j].gf\_multiply\_modular(multiply\_maxtrix[1][2], AES\_modulus, 8) ^ \  
 state\_array[3][j].gf\_multiply\_modular(multiply\_maxtrix[1][3], AES\_modulus, 8)  
  
 result[2][j] = state\_array[0][j].gf\_multiply\_modular(multiply\_maxtrix[2][0], AES\_modulus, 8) ^ \  
 state\_array[1][j].gf\_multiply\_modular(multiply\_maxtrix[2][1], AES\_modulus, 8) ^ \  
 state\_array[2][j].gf\_multiply\_modular(multiply\_maxtrix[2][2], AES\_modulus, 8) ^ \  
 state\_array[3][j].gf\_multiply\_modular(multiply\_maxtrix[2][3], AES\_modulus, 8)  
  
 result[3][j] = state\_array[0][j].gf\_multiply\_modular(multiply\_maxtrix[3][0], AES\_modulus, 8) ^ \  
 state\_array[1][j].gf\_multiply\_modular(multiply\_maxtrix[3][1], AES\_modulus, 8) ^ \  
 state\_array[2][j].gf\_multiply\_modular(multiply\_maxtrix[3][2], AES\_modulus, 8) ^ \  
 state\_array[3][j].gf\_multiply\_modular(multiply\_maxtrix[3][3], AES\_modulus, 8)  
 return result  
  
#######################  
  
def gen\_state\_array(last\_round\_array,bitvec):  
 for i in range(4):  
 for j in range(4):  
 last\_round\_array[j][i] = bitvec[32 \* i + 8 \* j: 32 \* i + 8 \* (j + 1)] # we want the value to be store in hex  
 return last\_round\_array  
#######################  
  
def encrypt():  
 #file\_to\_encryt = open(sys.argv[2], 'rb')  
 bv = BitVector(filename=sys.argv[2])  
 output = open(sys.argv[4],'w+')  
 key\_schedule, round\_keys = gen\_key()  
 s\_table = gen\_subbytes\_table()  
  
 #initialize the state array to all zero  
  
 state\_array = [[0 for x in range(4)] for x in range(4)]  
 last\_round\_array = [[0 for x in range(4)] for x in range(4)] # use for last round since we dont need to mix column  
 while (bv.more\_to\_read):  
 bitvec = bv.read\_bits\_from\_file( 128 )  
  
 if bitvec.length() % 128 is not 0:  
 bitvec = bitvec + BitVector(intVal=0, size=128 - bitvec.length() % 128)  
 bitvec ^= BitVector(hexstring=round\_keys[0]) #xor with the first round key  
  
  
 for i in range(1,14): # process only the first 13 round  
 state\_array = gen\_state\_array(last\_round\_array, bitvec)  
 state\_array = sub\_bytes(state\_array, s\_table)  
  
 state\_array = shift\_rows(state\_array)  
  
 state\_array = mix\_column(state\_array)  
  
 for k in range(4):  
 for j in range(4):  
 bitvec[32 \* k + 8 \* j: 32 \* k + 8 \* (j + 1)] = state\_array[j][k] # we want the value to be store in hex  
 bitvec ^= BitVector(hexstring=round\_keys[i])  
  
 state\_array = gen\_state\_array(last\_round\_array, bitvec)  
 state\_array = sub\_bytes(state\_array, s\_table)  
 state\_array = shift\_rows(state\_array)  
  
 for i in range(4):  
 for j in range(4):  
 bitvec[32 \* i + 8 \* j: 32 \* i + 8 \* (j + 1)] = state\_array[j][i] # we want the value to be store in hex  
 bitvec ^= BitVector(hexstring=round\_keys[14])  
  
 output.write(bitvec.get\_bitvector\_in\_hex())  
  
def decrypt():  
 file\_to\_decryt = open(sys.argv[2], 'r').read()  
 decrypted\_temp = open('temp.txt', 'w+b')  
 temp = BitVector(hexstring=file\_to\_decryt)  
 temp.write\_to\_file(decrypted\_temp)  
 decrypted\_temp.close()  
  
 bv = BitVector(filename='temp.txt')  
 output = open(sys.argv[4], 'w+b')  
 key\_schedule, round\_keys = gen\_key()  
 genTables()  
 os.remove('temp.txt')  
 #print(invSubBytesTable)  
  
 # initialize the state array to all zero  
  
 state\_array = [[0 for x in range(4)] for x in range(4)]  
 last\_round\_array = [[0 for x in range(4)] for x in range(4)] # use for last round since we dont need to mix column  
 while (bv.more\_to\_read):  
 bitvec = bv.read\_bits\_from\_file(128)  
  
 if bitvec.length() % 128 is not 0:  
 bitvec = bitvec + BitVector(intVal=0, size=128 - bitvec.length() % 128)  
 bitvec ^= BitVector(hexstring=round\_keys[14]) # xor with the first round key  
  
 for i in range(13, 0,-1): # process only the first 13 round  
 state\_array = gen\_state\_array(last\_round\_array, bitvec)  
  
 state\_array = inv\_shift\_rows(state\_array)  
  
 state\_array = sub\_bytes(state\_array, invSubBytesTable)  
  
 for k in range(4):  
 for j in range(4):  
 bitvec[32 \* k + 8 \* j: 32 \* k + 8 \* (j + 1)] = state\_array[j][k] # we want the value to be store in hex  
 bitvec ^= BitVector(hexstring=round\_keys[i])  
  
 state\_array = gen\_state\_array(last\_round\_array, bitvec)  
 state\_array = inv\_mix\_column(state\_array)  
  
 for x in range(4):  
 for y in range(4):  
 bitvec[32 \* x + 8 \* y: 32 \* x + 8 \* (y + 1)] = state\_array[y][x]  
  
 state\_array = inv\_shift\_rows(state\_array)  
 state\_array = sub\_bytes(state\_array, invSubBytesTable)  
 #state\_array = gen\_state\_array(last\_round\_array, bitvec)  
  
 for h in range(4):  
 for g in range(4):  
 bitvec[32 \* h + 8 \* g: 32 \* h + 8 \* (g + 1)] = state\_array[g][h] # we want the value to be store in hex  
 bitvec ^= BitVector(hexstring=round\_keys[0])  
  
 bitvec.write\_to\_file(output)  
  
  
 #### populate the state array  
  
 # we need to  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 if sys.argv[1] == '-e':  
 encrypt()  
 else:  
 decrypt()  
 #table = gen\_subbytes\_table()  
 #print(table)  
 #key\_schedule, round\_keys = gen\_key()  
 #print(type(round\_keys))