*"""  
Homework Number: 5  
Name: Sneha Mahapatra  
ECN Login: mahapat0  
Due Date: 02/25/2020  
"""*from sys import \*  
from BitVector import \*  
AES\_modulus = BitVector(bitstring='100011011')  
  
SUBBYTESTABLE = [99, 124, 119, 123, 242, 107, 111, 197, 48, 1, 103, 43, 254, 215, 171, 118, 202, 130, 201, 125, 250, 89, 71, 240,  
 173, 212, 162, 175, 156, 164, 114, 192, 183, 253, 147, 38, 54, 63, 247, 204, 52, 165, 229, 241, 113, 216, 49, 21,  
 4, 199, 35, 195, 24, 150, 5, 154, 7, 18, 128, 226, 235, 39, 178, 117, 9, 131, 44, 26, 27, 110, 90, 160, 82, 59,  
 214, 179, 41, 227, 47, 132, 83, 209, 0, 237, 32, 252, 177, 91, 106, 203, 190, 57, 74, 76, 88, 207, 208, 239, 170,  
 251, 67, 77, 51, 133, 69, 249, 2, 127, 80, 60, 159, 168, 81, 163, 64, 143, 146, 157, 56, 245, 188, 182, 218, 33,  
 16, 255, 243, 210, 205, 12, 19, 236, 95, 151, 68, 23, 196, 167, 126, 61, 100, 93, 25, 115, 96, 129, 79, 220, 34,  
 42, 144, 136, 70, 238, 184, 20, 222, 94, 11, 219, 224, 50, 58, 10, 73, 6, 36, 92, 194, 211, 172, 98, 145, 149, 228,  
 121, 231, 200, 55, 109, 141, 213, 78, 169, 108, 86, 244, 234, 101, 122, 174, 8, 186, 120, 37, 46, 28, 166, 180,  
 198, 232, 221, 116, 31, 75, 189, 139, 138, 112, 62, 181, 102, 72, 3, 246, 14, 97, 53, 87, 185, 134, 193, 29, 158,  
 225, 248, 152, 17, 105, 217, 142, 148, 155, 30, 135, 233, 206, 85, 40, 223, 140, 161, 137, 13, 191, 230, 66, 104,  
 65, 153, 45, 15, 176, 84, 187, 22] # for encryption  
KEY\_SCHEDULE = []  
SIZE = 256  
SIZEPLAIN = 128  
  
def encrypt(key, v0, image\_file, out\_file):  
 #Key is given to us  
 # Create Key Schedule  
 KEY\_SCHEDULE = keyInit(key)  
 first\_words = KEY\_SCHEDULE[0]  
  
 #Turn File into BitVector by Reading Image File  
 image = open(image\_file, 'rb')  
 imageno = image.readline()  
 height\_width = image.readline()  
 max\_pixVal = image.readline()  
 image.close()  
  
 #Open File for Encrypted Text  
 encryptedText = open(out\_file, "wb")  
 encryptedText.write(imageno)  
 encryptedText.write(height\_width)  
 encryptedText.write(max\_pixVal)  
 bv = BitVector(filename=image\_file)  
  
 index = -1  
 while(bv.more\_to\_read):  
  
 #Get 128 Bit for Plain Text and Ignore the header  
 if(index == -1):  
 XorAtEnd = bv.read\_bits\_from\_file(SIZEPLAIN-16)  
 XorAtEnd = bv.read\_bits\_from\_file(SIZEPLAIN)  
  
 #If block is < 128, pad zeroes  
 if (len(str(XorAtEnd)) % SIZEPLAIN != 0):  
 x = XorAtEnd.length() % SIZE  
 XorAtEnd.pad\_from\_right(SIZEPLAIN-x)  
  
 #THE BITVECTOR IS INTIALIZED BITVECTOR INCREMENT IF NECESSARY  
 index += 1  
 vX = v0.int\_val()  
 vX += index  
 vX = BitVector(intVal=vX)  
 bitvec = vX  
  
 # First Add Round Key  
 firstwordsHex = BitVector(hexstring = first\_words)  
 bitvec = bitvec ^ firstwordsHex  
 numRounds = 1  
  
 while(numRounds < 14):  
 #Turn Bit vector into State Array  
 statearray = matrixArray(bitvec, numRounds)  
  
 # SubBytes  
 statearray = substitution(statearray, 'E')  
  
 # ShiftRows  
 statearray = shiftRows(statearray, 'E')  
 #statearray = intToBitVector(statearray)  
  
 # Mix Columns --> Not on last round  
 statearray = mixColumns(statearray, 'E')  
 hexFinal = bitToHex(statearray)  
  
 # Add Round Keys  
 next\_word = KEY\_SCHEDULE[numRounds]  
 strX = ""  
 hexFinal = [''.join(x) for x in hexFinal]  
 hexFinal = BitVector(hexstring=strX.join(hexFinal))  
 next\_word = BitVector(hexstring=next\_word)  
 hexFinal = hexFinal ^ next\_word  
 bitvec = hexFinal  
 numRounds += 1  
 if(numRounds == 14):  
 #Turn Bit vector into State Array  
 statearray = matrixArray(bitvec, numRounds)  
  
 # SubBytes  
 statearray = substitution(statearray, 'E')  
  
 # ShiftRows  
 statearray = shiftRows(statearray, 'E')  
 #statearray = intToBitVector(statearray)  
 hexFinal = bitToHex(statearray)  
  
 #Add RoundKey  
 next\_word = KEY\_SCHEDULE[numRounds]  
 strX = ""  
 hexFinal = [''.join(x) for x in hexFinal]  
 hexFinal = BitVector(hexstring=strX.join(hexFinal))  
 next\_word = BitVector(hexstring=next\_word)  
 hexFinal = hexFinal ^ next\_word  
 #roundKeys(hexFinal, next\_word)  
 print(hexFinal.get\_bitvector\_in\_hex())  
 break  
 #XOR The plaintext with the encrypted File  
 hexFinal = hexFinal ^ XorAtEnd  
 hexFinal.write\_to\_file(encryptedText)  
 encryptedText.close()  
  
def intToBitVector(matrix):  
 for i in range(4):  
 for j in range(4):  
 matrix[i][j] = BitVector(intVal = matrix[i][j], size = 8)  
 return matrix  
  
def hextobit(matrix):  
 for i in range(4):  
 for j in range(4):  
 matrix[i][j] = BitVector(hexstring = matrix[i][j])  
 return matrix  
  
def bitToInt(matrix):  
 for i in range(4):  
 for j in range(4):  
 matrix[i][j] = int(matrix[i][j])  
 return matrix  
  
def bitToHex(matrix):  
 returnMatrix = [[0 for x in range(4)] for x in range(4)]  
 for i in range(4):  
 for j in range(4):  
 returnMatrix[i][j] = matrix[i][j].get\_bitvector\_in\_hex()  
 return returnMatrix  
  
def intToHex(matrix):  
 for i in range(4):  
 for j in range(4):  
 matrix[i][j] = format(matrix[i][j], 'x')  
 if(len(str(matrix[i][j])) == 1):  
 matrix[i][j] = '0' + matrix[i][j]  
 return matrix  
  
def matrixArray(bitvec, num):  
 bitvecHex = list(bitvec.get\_bitvector\_in\_hex())  
 statearray = [[0 for x in range(4)] for x in range(4)]  
 index = 0  
 for i in range(4):  
 for j in range(4):  
 statearray[i][j] = bitvecHex[index] + bitvecHex[index+1]  
 index += 2  
 if(num == 1):  
 statearray = [list(x) for x in zip(statearray[0], statearray[1], statearray[2], statearray[3])]  
 print(statearray)  
 return statearray  
  
def substitution(hexVector, charX):  
 for i in range(4):  
 for j in range(4):  
 hexVector[i][j] = BitVector(intVal = SUBBYTESTABLE[int(hexVector[i][j], 16)], size = 8)  
 return hexVector  
  
def shiftRows(vector, charX):  
 shift = 1  
 #vector = [list(x) for x in zip(vector[0], vector[1], vector[2], vector[3])]  
 while(shift < 4):  
 vector[shift] = rotateElemList(vector[shift], shift)  
 shift+=1  
 #vector = [list(x) for x in zip(vector[0], vector[1], vector[2], vector[3])]  
 return vector  
  
def rotateElemList(listX, shift):  
 return listX[shift:] + listX[:shift]  
  
def mixColumns(matrix, charX):  
 bv1 = BitVector(intVal=1, size=8)  
 bv2 = BitVector(intVal=2, size=8)  
 bv3 = BitVector(intVal=3, size=8)  
 #[[2, 3, 1, 1], [1, 2, 3, 1], [1, 1, 2, 3], [3, 1, 1, 2]]  
 mixColumns = [[bv2, bv3, bv1, bv1], [bv1, bv2, bv3, bv1], [bv1, bv1, bv2, bv3], [bv3, bv1, bv1, bv2]]  
 #mixColumns = [list(x) for x in zip(mixColumns[0], mixColumns[1], mixColumns[2], mixColumns[3])]  
 #matrix = [list(x) for x in zip(matrix[0], matrix[1], matrix[2], matrix[3])]  
 endMatrix = [[0 for x in range(4)] for x in range(4)]  
 for i in range(4):  
 for j in range(4):  
 bitvec0 = (mixColumns[i][0]).gf\_multiply\_modular(matrix[0][j], AES\_modulus, 8)  
 bitvec1 = (mixColumns[i][1]).gf\_multiply\_modular(matrix[1][j], AES\_modulus, 8)  
 bitvec2 = (mixColumns[i][2]).gf\_multiply\_modular(matrix[2][j], AES\_modulus, 8)  
 bitvec3 = (mixColumns[i][3]).gf\_multiply\_modular(matrix[3][j], AES\_modulus, 8)  
 endMatrix[i][j] = bitvec0 ^ bitvec1 ^ bitvec2 ^ bitvec3  
 #endMatrix = [list(x) for x in zip(endMatrix[0], endMatrix[1], endMatrix[2], endMatrix[3])]  
 return endMatrix  
  
def roundKeys(p, next\_words):  
 strX = ""  
 hexFinal = [''.join(x) for x in p]  
 hexFinal = BitVector(hexstring=strX.join(hexFinal))  
 next\_word = BitVector(hexstring=next\_words)  
 #hexFinal = hexFinal ^ next\_word  
 return hexFinal ^ next\_word  
  
def printKeySchedule(key\_schedule):  
 index = 0  
 for word in key\_schedule:  
 print("word " + str(index) + " " + str(word))  
 if((index+1) % 4 == 0):  
 print("\n")  
 index+=1  
  
def keyInit(key):  
 key\_words = []  
 key = key.strip()  
 key += '0' \* (SIZE // 8 - len(key)) if len(key) < SIZE // 8 else key[:SIZE // 8]  
 key\_bv = BitVector(textstring=key)  
 key\_words = gen\_key\_schedule\_256(key\_bv)  
  
 key\_schedule = []  
  
 enumkey\_word = enumerate(key\_words)  
 for word\_index, word in enumkey\_word:  
 keyword\_in\_ints = []  
 for i in range(4):  
 keyword\_in\_ints.append(word[i \* 8:i \* 8 + 8].intValue())  
 #print("word %d: %s" % (word\_index, str(keyword\_in\_ints)))  
 key\_schedule.append(keyword\_in\_ints)  
 #print(key\_schedule)  
 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 round\_keys  
  
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\_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 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))  
'''  
#Arguments:  
# iv: 128-bit initialization vector  
# image\_file: input .ppm image file name  
# out\_file: encrypted .ppm image file name  
# key\_file: String of file name containing encryption key (in ASCII)  
#Function Descrption:  
# Encrypts image\_file using CTR mode AES and writes said file to out\_file. No  
#required return value.  
def ctr\_aes\_image(iv, image\_file,out\_file, key\_file):  
 # Read Key  
 READKEY = open(key\_file, 'r')  
 key = READKEY.read()  
 READKEY.close()  
 encrypt(key, iv, image\_file, out\_file)  
 return 0