This program implements AES for encryption and decryption, which contains four basic steps: byte substitution, shifting rows, mixing columns, and adding keys. Part of the program comes from professor’s lecture notes

*# Homework Number: 4  
# Name: Zhengsen FU  
# ECN Login: fu216  
# Due Date: Feb 2/18  
# Python 3.7***import** sys  
**from** BitVector **import** \*  
  
mode = sys.argv[1] *# -e for encryption or -d for decryption*inputFileName = sys.argv[2] *# message.txt or encrypted.txt*keyFileName = sys.argv[3] *# key.txt*outputFileName = sys.argv[4] *# encrypted.txt or decrypted.txt*AES\_modulus = BitVector(bitstring=**'100011011'**)  
subBytesTable = [] *# for encryption*invSubBytesTable = [] *# for decryption***def** genTables():  
 *# this function is from lecture note* 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\_256(key\_bv):  
 *# this function comes from lecture note* byte\_sub\_table = subBytesTable  
 *# 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** get\_key\_from\_user():  
 *# this function is from lecture note* keysize = 256  
 *# if sys.version\_info[0] == 3:  
 # keysize = int(input("\nAES Key size: "))  
 # assert any(x == keysize for x in [128,192,256]), \  
 # "keysize is wrong (must be one of 128, 192, or 256) --- aborting"  
 # key = input("\nEnter key (any number of chars): ")  
 # else:  
 # keysize = int(raw\_input("\nAES Key size: "))  
 # assert any(x == keysize for x in [128,192,256]), \  
 # "keysize is wrong (must be one of 128, 192, or 256) --- aborting"  
 # key = raw\_input("\nEnter key (any number of chars): ")* **with** open(keyFileName) **as** keyFile:  
 key = keyFile.read()  
 key = key.strip()  
 key += **'0'** \* (keysize // 8 - len(key)) **if** len(key) < keysize // 8 **else** key[:keysize // 8]  
 key\_bv = BitVector(textstring=key)  
 **return** keysize, key\_bv  
  
  
**def** encryption():  
 *# generate round keys* keysize, key\_bv = get\_key\_from\_user()  
 key\_words = gen\_key\_schedule\_256(key\_bv)  
 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())  
 *# if word\_index % 4 == 0:  
 # print("\n")  
 # print("word %d: %s" % (word\_index, str(keyword\_in\_ints)))* key\_schedule.append(keyword\_in\_ints)  
 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()  
  
 input\_bv = BitVector(filename=inputFileName)  
 output\_bv = BitVector(size=0)  
 *# begin the encryption process* **while** input\_bv.more\_to\_read:  
 bitVec = input\_bv.read\_bits\_from\_file(128)  
 **if** bitVec.size != 128:  
 bitVec.pad\_from\_right(128 - bitVec.size)  
 *# add first round key* bitVec ^= BitVector(hexstring=round\_keys[0])  
 stateArr = generateStateArray(bitVec)  
  
 *# each round except last one* **for** round\_number **in** range(0, num\_rounds - 1):  
 *# byte substitution* stateArr = byteSub(stateArr, subBytesTable)  
 *# shift rows* stateArr = shiftRow(stateArr)  
 *# mix column* stateArr = mixColumn(stateArr)  
 *# add round key* bitVec = blockFromStateArr(stateArr)  
 bitVec ^= BitVector(hexstring=round\_keys[round\_number + 1])  
 stateArr = generateStateArray(bitVec)  
  
 *# last round  
 # byte substitution* stateArr = byteSub(stateArr, subBytesTable)  
 *# shift rows* stateArr = shiftRow(stateArr)  
 *# add round key* bitVec = blockFromStateArr(stateArr)  
 bitVec ^= BitVector(hexstring=round\_keys[num\_rounds])  
 output\_bv += bitVec  
 outputHexString = output\_bv.get\_hex\_string\_from\_bitvector()  
 **with** open(outputFileName, **'w'**) **as** outFile:  
 outFile.write(outputHexString)  
 **return  
  
  
def** decryption():  
 *# generate round keys* keysize, key\_bv = get\_key\_from\_user()  
 key\_words = gen\_key\_schedule\_256(key\_bv)  
 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())  
 *# if word\_index % 4 == 0:  
 # print("\n")  
 # print("word %d: %s" % (word\_index, str(keyword\_in\_ints)))* key\_schedule.append(keyword\_in\_ints)  
 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()  
  
 *# read from encryption file* **with** open(inputFileName) **as** inputFile:  
 hexString = inputFile.read()  
 inputBitVec = BitVector(hexstring=hexString) *# all bits from file* counter = 0 *# counter for blocks for going through* output\_bv = BitVector(size=0)  
 boundary = inputBitVec.size // 128 *# number of blocks* **if** inputBitVec.size % 128: *# if there are bit vector containing less than 64 bits* inputBitVec.pad\_from\_right(128 - (inputBitVec.size % 128)) *# pad 0 to the end if last block does not have enough bits* boundary += 1  
 *# begin decryption* **while** counter < boundary:  
 bitVec = inputBitVec[counter \* 128: counter \* 128 + 128]  
 *# add first round key* bitVec ^= BitVector(hexstring=round\_keys[num\_rounds])  
 stateArr = generateStateArray(bitVec)  
  
 *# each round except last one* **for** round\_number **in** range(0, num\_rounds - 1):  
 *# Inverse shift rows* stateArr = invShiftRow(stateArr)  
 *# Inverse substitute bytes* stateArr = byteSub(stateArr, invSubBytesTable)  
 *# add round key* bitVec = blockFromStateArr(stateArr)  
 bitVec ^= BitVector(hexstring=round\_keys[num\_rounds - 1 - round\_number])  
 stateArr = generateStateArray(bitVec)  
 *# Inverse mix column* stateArr = invMixColumn(stateArr)  
  
 *# last round  
 # Inverse shift rows* stateArr = invShiftRow(stateArr)  
 *# Inverse substitute bytes* stateArr = byteSub(stateArr, invSubBytesTable)  
 *# add round key* bitVec = blockFromStateArr(stateArr)  
 bitVec ^= BitVector(hexstring=round\_keys[0])  
 output\_bv += bitVec  
 counter += 1  
 outputString = output\_bv.get\_bitvector\_in\_ascii()  
 **with** open(outputFileName, **'w'**) **as** outFile:  
 outFile.write(outputString)  
 **return  
  
  
def** invMixColumn(stateArr):  
 newArr = [[**None for** \_ **in** range(4)] **for** \_ **in** range(4)]  
 timesE = BitVector(hexstring=**'0e'**) *# used for GF(2^8) multiplication* timesB = BitVector(hexstring=**'0b'**) *# used for GF(2^8) multiplication* timesD = BitVector(hexstring=**'0d'**) *# used for GF(2^8) multiplication* times9 = BitVector(hexstring=**'09'**) *# used for GF(2^8) multiplication* modulus = BitVector(bitstring=**'100011011'**)  
 **for** column **in** range(4):  
 *# first row* newArr[0][column] = timesE.gf\_multiply\_modular(stateArr[0][column], modulus, 8)  
 newArr[0][column] ^= timesB.gf\_multiply\_modular(stateArr[1][column], modulus, 8)  
 newArr[0][column] ^= timesD.gf\_multiply\_modular(stateArr[2][column], modulus, 8)  
 newArr[0][column] ^= times9.gf\_multiply\_modular(stateArr[3][column], modulus, 8)  
 *# second row* newArr[1][column] = times9.gf\_multiply\_modular(stateArr[0][column], modulus, 8)  
 newArr[1][column] ^= timesE.gf\_multiply\_modular(stateArr[1][column], modulus, 8)  
 newArr[1][column] ^= timesB.gf\_multiply\_modular(stateArr[2][column], modulus, 8)  
 newArr[1][column] ^= timesD.gf\_multiply\_modular(stateArr[3][column], modulus, 8)  
 *# third row* newArr[2][column] = timesD.gf\_multiply\_modular(stateArr[0][column], modulus, 8)  
 newArr[2][column] ^= times9.gf\_multiply\_modular(stateArr[1][column], modulus, 8)  
 newArr[2][column] ^= timesE.gf\_multiply\_modular(stateArr[2][column], modulus, 8)  
 newArr[2][column] ^= timesB.gf\_multiply\_modular(stateArr[3][column], modulus, 8)  
 *# forth row* newArr[3][column] = timesB.gf\_multiply\_modular(stateArr[0][column], modulus, 8)  
 newArr[3][column] ^= timesD.gf\_multiply\_modular(stateArr[1][column], modulus, 8)  
 newArr[3][column] ^= times9.gf\_multiply\_modular(stateArr[2][column], modulus, 8)  
 newArr[3][column] ^= timesE.gf\_multiply\_modular(stateArr[3][column], modulus, 8)  
 **return** newArr  
  
  
**def** invShiftRow(stateArr):  
 newArr = [[stateArr[0][0], stateArr[0][1], stateArr[0][2], stateArr[0][3]],  
 [stateArr[1][3], stateArr[1][0], stateArr[1][1], stateArr[1][2]],  
 [stateArr[2][2], stateArr[2][3], stateArr[2][0], stateArr[2][1]],  
 [stateArr[3][1], stateArr[3][2], stateArr[3][3], stateArr[3][0]]]  
 **return** newArr  
  
  
**def** mixColumn(stateArr):  
 newArr = [[**None for** \_ **in** range(4)] **for** \_ **in** range(4)]  
 times2 = BitVector(hexstring=**'02'**) *# used for GF(2^8) multiplication* times3 = BitVector(hexstring=**'03'**) *# used for GF(2^8) multiplication* modulus = BitVector(bitstring=**'100011011'**)  
 **for** column **in** range(4):  
 *# first row* newArr[0][column] = times2.gf\_multiply\_modular(stateArr[0][column], modulus, 8)  
 newArr[0][column] ^= times3.gf\_multiply\_modular(stateArr[1][column], modulus, 8)  
 newArr[0][column] ^= stateArr[2][column]  
 newArr[0][column] ^= stateArr[3][column]  
 *# second row* newArr[1][column] = stateArr[0][column]  
 newArr[1][column] ^= times2.gf\_multiply\_modular(stateArr[1][column], modulus, 8)  
 newArr[1][column] ^= times3.gf\_multiply\_modular(stateArr[2][column], modulus, 8)  
 newArr[1][column] ^= stateArr[3][column]  
 *# third row* newArr[2][column] = stateArr[0][column]  
 newArr[2][column] ^= stateArr[1][column]  
 newArr[2][column] ^= times2.gf\_multiply\_modular(stateArr[2][column], modulus, 8)  
 newArr[2][column] ^= times3.gf\_multiply\_modular(stateArr[3][column], modulus, 8)  
 *# forth row* newArr[3][column] = times3.gf\_multiply\_modular(stateArr[0][column], modulus, 8)  
 newArr[3][column] ^= stateArr[1][column]  
 newArr[3][column] ^= stateArr[2][column]  
 newArr[3][column] ^= times2.gf\_multiply\_modular(stateArr[3][column], modulus, 8)  
 **return** newArr  
  
  
**def** shiftRow(stateArr):  
 newArr = [[stateArr[0][0], stateArr[0][1], stateArr[0][2], stateArr[0][3]],  
 [stateArr[1][1], stateArr[1][2], stateArr[1][3], stateArr[1][0]],  
 [stateArr[2][2], stateArr[2][3], stateArr[2][0], stateArr[2][1]],  
 [stateArr[3][3], stateArr[3][0], stateArr[3][1], stateArr[3][2]]]  
 **return** newArr  
  
  
**def** blockFromStateArr(stateArr):  
 *# this function generate a 128 bit block from a state arr* bitVec = BitVector(size=0)  
 **for** column **in** range(4):  
 **for** row **in** range(4):  
 bitVec += stateArr[row][column]  
 **return** bitVec  
  
  
**def** byteSub(stateArr, subTable):  
 *# this function uses subTable to substitute each byte un state arr* **for** row **in** range(4):  
 **for** column **in** range(4):  
 stateArr[row][column] = BitVector(intVal=subTable[stateArr[row][column].intValue()], size=8)  
 **return** stateArr  
  
  
**def** generateStateArray(bitVec):  
 *# generates State array form a bit vector* stateArr = [[**None for** \_ **in** range(4)] **for** \_ **in** range(4)]  
 **for** column **in** range(0, 4):  
 **for** row **in** range(0, 4):  
 byteNum = column \* 4 + row  
 stateArr[row][column] = bitVec[byteNum \* 8: (byteNum + 1) \* 8]  
 **return** stateArr  
  
  
genTables()  
**if** mode == **'-e'**:  
 encryption()  
**elif** mode == **'-d'**:  
 decryption()  
**else**:  
 print(**"Incorrect mode input"**)

Output of encryption:



Output of decryption:

This is an unusual paragraph. I'm curious how quickly you can find out what is so unusual about it? It looks so plain you would think nothing was wrong with it! In fact, nothing is wrong with it! It is unusual though. Study it, and think about it, but you still may not find anything odd. But if you work at it a bit, you might find out! Try to do so without any coaching! You most probably won't, at first, find anything particularly odd or unusual or in any way dissimilar to any ordinary composition. That is not at all surprising, for it is no strain to accomplish in so short a paragraph a stunt similar to that which an author did throughout all of his book, without spoiling a good writing job, and it was no small book at that.