Script:

import sys

import io

import argparse

import numpy as np

from BitVector import \*

def genTables(cond):

AES\_modulus = BitVector(bitstring="100011011")

c = BitVector(bitstring="001100011")

d = BitVector(bitstring="00000101")

subBytesTable = [] # SBox for encryption

invSubBytesTable = []

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))

if (cond):

return subBytesTable

else:

return invSubBytesTable

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 = []

AES\_modulus = BitVector(bitstring="100011011")

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 gee(keyword, round\_constant, byte\_sub\_table):

AES\_modulus = BitVector(bitstring="100011011")

## print(keyword,"here")

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 shiftrowleft(statearray):

temp = [0,0,0,0]

flag = 0

for i in range(1,4):

while(flag<i):

statearray[0][i],statearray[1][i],statearray[2][i],statearray[3][i] = leftRotate([statearray[0][i],statearray[1][i],statearray[2][i],statearray[3][i]],4)

flag=flag+1

flag=0

return statearray

def leftRotate(line,n):

temp = line[0]

for i in range(n-1):

line[i] = line [i+1]

line[n-1] = temp

return line

def shiftrowright(statearray):

temp = [0,0,0,0]

flag = 0

for i in range(1,4):

while(flag<i):

statearray[0][i],statearray[1][i],statearray[2][i],statearray[3][i] = rightRotate([statearray[0][i],statearray[1][i],statearray[2][i],statearray[3][i]],4)

flag=flag+1

flag=0

return statearray

def rightRotate(line,n):

temp = line[3]

for i in range(n-1):

line[3-i] = line [2-i]

line[0] = temp

return line

def mixcolumn(inputarray,cond):

statearray = [[0 for x in range(4)] for x in range(4)]

AES = BitVector(bitstring='100011011')

if(cond):

two = BitVector(bitstring = '00000010')

thr = BitVector(bitstring = '00000011')

for i in range(4):

fir\_col = inputarray[i][0].gf\_multiply\_modular(two,AES,8)

sec\_col = inputarray[i][1].gf\_multiply\_modular(thr,AES,8)

fir\_col\_1 = inputarray[i][1].gf\_multiply\_modular(two,AES,8)

sec\_col\_2 = inputarray[i][2].gf\_multiply\_modular(thr,AES,8)

fir\_col\_3 = inputarray[i][2].gf\_multiply\_modular(two,AES,8)

sec\_col\_4 = inputarray[i][3].gf\_multiply\_modular(thr,AES,8)

fir\_col\_5 = inputarray[i][3].gf\_multiply\_modular(two,AES,8)

sec\_col\_6 = inputarray[i][0].gf\_multiply\_modular(thr,AES,8)

statearray[i][0] = fir\_col ^ sec\_col ^ inputarray[i][2] ^ inputarray[i][3]

statearray[i][1] = fir\_col\_1 ^ sec\_col\_2 ^ inputarray[i][0] ^ inputarray[i][3]

statearray[i][2] = fir\_col\_3 ^ sec\_col\_4 ^ inputarray[i][1] ^ inputarray[i][0]

statearray[i][3] = fir\_col\_5 ^ sec\_col\_6 ^ inputarray[i][2] ^ inputarray[i][1]

return statearray

else:

zB = BitVector(bitstring='00001011')

zE = BitVector(bitstring='00001110')

zD = BitVector(bitstring='00001101')

z9 = BitVector(bitstring='00001001')

for i in range(4):

statearray[i][0] = (inputarray[i][0].gf\_multiply\_modular(zE,AES,8)) ^ \

(inputarray[i][1].gf\_multiply\_modular(zB,AES,8)) ^ \

(inputarray[i][2].gf\_multiply\_modular(zD,AES,8)) ^ \

(inputarray[i][3].gf\_multiply\_modular(z9,AES,8))

statearray[i][1] = (inputarray[i][1].gf\_multiply\_modular(zE,AES,8)) ^ \

(inputarray[i][2].gf\_multiply\_modular(zB,AES,8)) ^ \

(inputarray[i][3].gf\_multiply\_modular(zD,AES,8)) ^ \

(inputarray[i][0].gf\_multiply\_modular(z9,AES,8))

statearray[i][2] = (inputarray[i][2].gf\_multiply\_modular(zE,AES,8)) ^ \

(inputarray[i][3].gf\_multiply\_modular(zB,AES,8)) ^ \

(inputarray[i][0].gf\_multiply\_modular(zD,AES,8)) ^ \

(inputarray[i][1].gf\_multiply\_modular(z9,AES,8))

statearray[i][3] = (inputarray[i][3].gf\_multiply\_modular(zE,AES,8)) ^ \

(inputarray[i][0].gf\_multiply\_modular(zB,AES,8)) ^ \

(inputarray[i][1].gf\_multiply\_modular(zD,AES,8)) ^ \

(inputarray[i][2].gf\_multiply\_modular(z9,AES,8))

return statearray

def encrypt(message,key,encrypted):

f\_key = open(key,"r")

key = BitVector(textstring=f\_key.read())

bv = BitVector(filename=message)

a = []

round\_keys = [None for i in range(15)]

key\_words = gen\_key\_schedule\_256(key)

subBytesTable = genTables(True)

statearray = [[0 for x in range(4)] for x in range(4)]

for i in range(15):

round\_keys[i] = (key\_words[i\*4] + key\_words[i\*4+1] + key\_words[i\*4+2] + key\_words[i\*4+3])

## print(round\_keys[0])

with open(encrypted,"w") as f:

while (bv.more\_to\_read):

bitvec = bv.read\_bits\_from\_file(128)

if (len(bitvec)<128):

bitvec.pad\_from\_right(128-len(bitvec))

#first round xor with key##################

bitvec = bitvec ^ (round\_keys[0])

for k in range(14):

# substitution bytes

for i in range(16):

L,R = bitvec[i\*8:(8+i\*8)].divide\_into\_two()

bitvec[i\*8:(8+i\*8)] = BitVector(intVal=subBytesTable[int(L) \* 16 + int(R)], size=8)

z=0

for j in range(4):

for i in range(4):

statearray[j][i] = bitvec[z:z+8]

z = z+8

# shift rows

statearray = shiftrowleft(statearray)

# Mix Column

if (k != 13):

statearray = mixcolumn(statearray,True)

bitvec = BitVector(size = 0)

for j in range(4):

for i in range(4):

bitvec = bitvec + statearray[j][i]

z = z+8

#Add roundkey

bitvec = bitvec ^ (round\_keys[k+1])

f.write(bitvec.get\_bitvector\_in\_hex())#hexstring

f.close()

return

def decrypt(encrypted,key,decrypted):

f\_key = open(key,"r")

key\_s = BitVector(textstring=f\_key.read())

round\_keys = [None for i in range(15)]

f\_input = open(encrypted,"r")

bv = BitVector( hexstring = f\_input.read())

invsubBytesTable = genTables(False)

key\_words = gen\_key\_schedule\_256(key\_s)

statearray = [[0 for x in range(4)] for x in range(4)]

for i in range(15):

round\_keys[i] = (key\_words[i\*4] + key\_words[i\*4+1] + key\_words[i\*4+2] + key\_words[i\*4+3])

round\_keys.reverse()

statearray = [[0 for x in range(4)] for x in range(4)]

m=0

with open(decrypted,"wb") as f:

for l in range(int(len(bv)/128)):

bitvec = bv[m:m+128]

m = m+128

#first round xor with key##################

bitvec = bitvec ^ (round\_keys[0])

for k in range(14):

z=0

for j in range(4):

for i in range(4):

statearray[j][i] = bitvec[z:z+8]

z = z+8

# Invshift rows

statearray = shiftrowright(statearray)

# Invsubstitution bytes

for i in range(4):

for j in range(4):

statearray[i][j] = BitVector(intVal=invsubBytesTable[int(statearray[i][j])],size=8)

#Add roundkey

bitvec = BitVector(size = 0)

for j in range(4):

for i in range(4):

bitvec = bitvec + statearray[j][i]

z = z+8

bitvec = bitvec ^ (round\_keys[k+1])

z=0

for j in range(4):

for i in range(4):

statearray[j][i] = bitvec[z:z+8]

z = z+8

# Mix Column

if (k != 13):

statearray = mixcolumn(statearray,False)

bitvec = BitVector(size = 0)

for j in range(4):

for i in range(4):

bitvec = bitvec + statearray[j][i]

z = z+8

bitvec.write\_to\_file(f)#hexstring

f.close()

return

if \_\_name\_\_ == '\_\_main\_\_':

my\_parser = argparse.ArgumentParser()

my\_parser.add\_argument('-d',"--d",type=str,nargs='\*', default=None)

my\_parser.add\_argument('-e',"--e", type=str,nargs='\*', default=None)

args = my\_parser.parse\_args()

if args.e:

encrypt('message.txt', 'key.txt', 'encrypted.txt')

if args.d:

decrypt('encrypted.txt', 'key.txt', 'decrypted.txt')