Homework Number: 05
Name: Yi Qiao
ECN Login: qiao22
Due Date: 02/19/2019

Encrypted Image

Code

Part 1

```
#! /usr/bin/env python3
from BitVector import *
from math import ceil
import time
import sys
import copy

AES_modulus = BitVector(bitstring='100011011')
subBytesTable = []                                                  # for encryption
MAXROUND = 14

v0=BitVector(textstring="computersecurity")

def genTables():
    # Credit to: Avi Kak (February 15, 2015)
    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))


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 = 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]
```

```python
        else:
            sys.exit("error in key scheduling algo for i = %d" % i)
    return key_words


def get_key_from_user(key_file="key.txt"):
    # only get 256 bits key in this case
    bvFp = BitVector(filename=key_file)
    key_bv = bvFp.read_bits_from_file(256)
    return key_bv


def stateBlock_to_bitvector(state):
    # reform into bitvector
    bv = BitVector(size=0)
    for j in range(4):
        for i in range(4):
            bv += state[i][j]
    return bv


def bitvector_to_stateBlock(bv):
    return [[bv[j*32+i*8:j*32+i*8+8] for j in range(4)] for i in range(4)]


def AESEncryptOneBlock(bv, key_bytes):

    def oneRoundAESEncrypt(bv, roundkey, lastround=False):

        # reform into 4*4 block
        state = bitvector_to_stateBlock(bv)

        # substitution
        for i in range(4):
            for j in range(4):
                state[i][j] = BitVector(intVal=subBytesTable[int(state[i][j])],
                 ↪  size=8)

        # shift row
        for i in range(1,4):
            state[i] = state[i][i:] + state[i][:i]

        if not lastround:
            # mix columns
            state_cpy = copy.deepcopy(state)
            factor = [BitVector(hexstring="02"), BitVector(hexstring="03"),
             ↪  BitVector(hexstring="01"), BitVector(hexstring="01")]
            for j in range(4):
```

```
                for i in range(4):
                    output = BitVector(size=8)
                    for k in range(4):
                        output ^=
                        ↪  state_cpy[k][j].gf_multiply_modular(factor[k-i],AES_modulus,8)
                    state[i][j] = output

        # reform into bitvector
        bv = stateBlock_to_bitvector(state)

        # Add round key
        bv ^= roundkey

        return bv

    if bv.length() < 128:
        bv.pad_from_right(128-bv.length())

    # add round key
    key = key_bytes[0] + key_bytes[1] + key_bytes[2] + key_bytes[3]
    bv ^= key

    # encrypt 14 rounds
    for i in range(MAXROUND):
        key = key_bytes[(i+1)*4] + key_bytes[(i+1)*4+1] + key_bytes[(i+1)*4+2] +
        ↪  key_bytes[(i+1)*4+3]
        bv = oneRoundAESEncrypt(bv, key, i == MAXROUND-1)

    return bv


def x931(v0, dt, totalNum, key_file='key.txt'):

    v_last = v0
    key_bv = get_key_from_user(key_file)
    key_bytes = gen_key_schedule_256(key_bv)

    encryptedTime = AESEncryptOneBlock(dt, key_bytes)
    ranNums = [None for _ in range(totalNum)]
    for i in range(totalNum):
        ranNums[i] = AESEncryptOneBlock(encryptedTime^v_last, key_bytes)
        v_last = AESEncryptOneBlock(ranNums[i] ^ encryptedTime, key_bytes)

    return [int(x) for x in ranNums]

if __name__ == "__main__":
    genTables()
```

```
    dt =
     ↪  BitVector(intVal=int(10**6*time.time()))+BitVector(intVal=int(10**6*time.time()))
    rans = x931(v0,dt,10)
    for ran in rans:
        print(ran)
```

Part 2

```
#! /usr/bin/env python3
from BitVector import *
from math import ceil
import time
import sys
import copy

AES_modulus = BitVector(bitstring='100011011')
subBytesTable = []                                                      # for
 ↪  encryption
MAXROUND = 14

v0=BitVector(textstring="computersecurity")

def genTables():
    # Credit to: Avi Kak (February 15, 2015)
    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))


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 = 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(key_file="key.txt"):
    # only get 256 bits key in this case
    bvFp = BitVector(filename=key_file)
    key_bv = bvFp.read_bits_from_file(256)
    return key_bv


def stateBlock_to_bitvector(state):
    # reform into bitvector
```

```
    bv = BitVector(size=0)
    for j in range(4):
        for i in range(4):
            bv += state[i][j]
    return bv


def bitvector_to_stateBlock(bv):
    return [[bv[j*32+i*8:j*32+i*8+8] for j in range(4)] for i in range(4)]


def AESEncryptOneBlock(bv, key_bytes):

    def oneRoundAESEncrypt(bv, roundkey, lastround=False):

        # reform into 4*4 block
        state = bitvector_to_stateBlock(bv)

        # substitution
        for i in range(4):
            for j in range(4):
                state[i][j] = BitVector(intVal=subBytesTable[int(state[i][j])],
                ↪  size=8)

        # shift row
        for i in range(1,4):
            state[i] = state[i][i:] + state[i][:i]

        if not lastround:
            # mix columns
            state_cpy = copy.deepcopy(state)
            factor = [BitVector(hexstring="02"), BitVector(hexstring="03"),
            ↪  BitVector(hexstring="01"), BitVector(hexstring="01")]
            for j in range(4):
                for i in range(4):
                    output = BitVector(size=8)
                    for k in range(4):
                        output ^=
                        ↪  state_cpy[k][j].gf_multiply_modular(factor[k-i],AES_modulus,8)
                    state[i][j] = output

        # reform into bitvector
        bv = stateBlock_to_bitvector(state)

        # Add round key
        bv ^= roundkey

        return bv
```

```python
    if bv.length() < 128:
        bv.pad_from_right(128-bv.length())

    # add round key
    key = key_bytes[0] + key_bytes[1] + key_bytes[2] + key_bytes[3]
    bv ^= key

    # encrypt 14 rounds
    for i in range(MAXROUND):
        key = key_bytes[(i+1)*4] + key_bytes[(i+1)*4+1] + key_bytes[(i+1)*4+2] +
        ↪   key_bytes[(i+1)*4+3]
        bv = oneRoundAESEncrypt(bv, key, i == MAXROUND-1)

    return bv


def x931(v0, dt, totalNum, key_file='key.txt'):

    v_last = v0
    key_bv = get_key_from_user(key_file)
    key_bytes = gen_key_schedule_256(key_bv)
    encryptedTime = AESEncryptOneBlock(dt, key_bytes)
    ranNums = [None for _ in range(totalNum)]
    for i in range(totalNum):
        ranNums[i] = AESEncryptOneBlock(encryptedTime^v_last, key_bytes)
        v_last = AESEncryptOneBlock(ranNums[i] ^ encryptedTime, key_bytes)

    return [int(x) for x in ranNums]


def ctr_aes_image(iv, image_file='image.ppm', out_file='enc_image.ppm',
↪   key_file='key.txt'):
    header = b""
    content = b""
    with open(image_file, "rb") as fp:
        for i in range(3):
            header += fp.readline()
        content = fp.read()

    content_bv = BitVector(rawbytes=content)
    key_bv = get_key_from_user(key_file)
    key_bytes = gen_key_schedule_256(key_bv)
    with open(out_file, "wb") as fp:
        fp.write(header)
        for i in range(0,len(content_bv),128):
            print("block: %d / %d"%(i, len(content_bv)))
            iv+=1
```

```
            encrypted_bv = AESEncryptOneBlock(BitVector(intVal = iv, size = 128),
            ↪  key_bytes)
            img_blk = content_bv[i:min(len(content_bv), i+128)]
            if len(img_blk) < 128:
                img_blk.pad_from_right(128-len(img_blk))
            encrypted_img_blk = img_blk ^ encrypted_bv
            encrypted_img_blk.write_to_file(fp)

    print("img encrypted")


if __name__ == "__main__":
    genTables()
    print("AES Table generated")
    dt =
    ↪  BitVector(intVal=int(10**6*time.time()))+BitVector(intVal=int(10**6*time.time()))
    ctr_aes_image(x931(v0,dt,1,'key.txt')[0],key_file='key2.txt')
```