

```

1  #!/usr/bin/env/python3
2  # Homework Number: hw04
3  # Name: Shu Hwai Teoh
4  # ECN Login: teoh0
5  # Due Date: Tuesday 2/18/2020 at 4:29PM
6  import sys
7  from BitVector import *
8
9  AES_modulus = BitVector(bitstring='100011011')
10 subBytesTable = [] # for encryption
11 invSubBytesTable = [] # for decryption
12
13 def genTables():
14     c = BitVector(bitstring='01100011')
15     d = BitVector(bitstring='00000101')
16     for i in range(0, 256):
17         # For the encryption SBox, find the multiplicative inverse x' = x_in^(-1) in
18         # GF(2^8)
19         a = BitVector(intVal = i, size=8).gf_MI(AES_modulus, 8) if i != 0 else
20         BitVector(intVal=0)
21         # For bit scrambling for the encryption SBox entries:
22         # scramble the bits of x' by XORing x' with
23         # four different circularly rotated versions of itself
24         # and with a special constant byte c = 0x63.
25         # The four circular rotations are through 4, 5, 6, and 7 bit positions to
26         # the right.
27         a1,a2,a3,a4 = [a.deep_copy() for x in range(4)]
28         a ^= (a1 >> 4) ^ (a2 >> 5) ^ (a3 >> 6) ^ (a4 >> 7) ^ c
29         subBytesTable.append(int(a))
30         # For the decryption Sbox:
31         b = BitVector(intVal = i, size=8)
32         # For bit scrambling for the decryption SBox entries:
33         b1,b2,b3 = [b.deep_copy() for x in range(3)]
34         b = (b1 >> 2) ^ (b2 >> 5) ^ (b3 >> 7) ^ d
35         check = b.gf_MI(AES_modulus, 8)
36         b = check if isinstance(check, BitVector) else 0
37         invSubBytesTable.append(int(b))
38
39 def gen_key_schedule_256(key_bv):
40     # byte_sub_table = gen_subbytes_table()
41     # We need 60 keywords (each keyword consists of 32 bits) in the key schedule for
42     # 256 bit AES. The 256-bit AES uses the first four keywords to xor the input
43     # block with. Subsequently, each of the 14 rounds uses 4 keywords from the key
44     # schedule. We will store all 60 keywords in the following list:
45     key_words = [None for i in range(60)]
46     round_constant = BitVector(intVal = 0x01, size=8)
47     for i in range(8):
48         key_words[i] = key_bv[i*32 : i*32 + 32]
49     for i in range(8,60):
50         if i%8 == 0:
51             kwd, round_constant = gee(key_words[i-1], round_constant, subBytesTable)
52             key_words[i] = key_words[i-8] ^ kwd
53         elif (i - (i//8)*8) < 4:
54             key_words[i] = key_words[i-8] ^ key_words[i-1]
55         elif (i - (i//8)*8) == 4:
56             key_words[i] = BitVector(size = 0)
57             for j in range(4):
58                 key_words[i] += BitVector(intVal =
59
60                                     subBytesTable[key_words[i-1][8*j:8*j+8].intValue()],
61                                     size = 8)
62             key_words[i] ^= key_words[i-8]
63         elif ((i - (i//8)*8) > 4) and ((i - (i//8)*8) < 8):
64             key_words[i] = key_words[i-8] ^ key_words[i-1]
65         else:
66             sys.exit("error in key scheduling algo for i = %d" % i)
67     return key_words
68
69 def gee(keyword, round_constant, byte_sub_table):
70     '''
71     This is the g() function for key expansion.
72     '''

```

```

68     rotated_word = keyword.deep_copy()
69     rotated_word << 8
70     newword = BitVector(size = 0)
71     for i in range(4):
72         newword += BitVector(intVal =
            byte_sub_table[rotated_word[8*i:8*i+8].intValue()], size = 8)
73     newword[:8] ^= round_constant
74     round_constant = round_constant.gf_multiply_modular(BitVector(intVal = 0x02),
        AES_modulus, 8)
75     return newword, round_constant
76
77 def keyEncryptExpend():
78     # read key string from key.txt and turn it into a bitVector
79     with open(sys.argv[3], "r") as f:
80         key = f.read().strip()
81     key_bv = BitVector(textstring=key)
82     key_words = gen_key_schedule_256(key_bv)
83     key_schedule = []
84     #Each 32-bit word of the key schedule is shown as a sequence of 4 one-byte
    integers
85     for word_index, word in enumerate(key_words):
86         keyword_in_ints = []
87         for i in range(4):
88             keyword_in_ints.append(word[i*8:i*8+8].intValue())
89         # if word_index % 4 == 0: print("\n")
90         # print("word %d:  %s" % (word_index, str(keyword_in_ints)))
91         key_schedule.append(keyword_in_ints)
92     num_rounds = 14
93     round_keys = [None for i in range(num_rounds+1)]
94     # de_round_key = [None for i in range(num_rounds+1)]
95     for i in range(num_rounds+1):
96         round_keys[i] = (key_words[i*4] + key_words[i*4+1] + key_words[i*4+2] +
97             key_words[i*4+3]).get_bitvector_in_hex()
98         # de_round_key[num_rounds-i] = key_words[i*4+3] + key_words[i*4+2] +
99         key_words[i*4+1] + key_words[i*4]
100
101     return round_keys #, de_round_key #list of 32-bit bitVector (each round key has
    4 words)
102
103 def AES_Encrypt(fileName, round_keys):
104     FILEIN = open(fileName)
105     input_bv = BitVector(textstring=FILEIN.read())
106     # create empty bit vector to store output
107     output_bv = BitVector(size=0)
108     # loop through all the input and extract 64 bit at a time
109     for j in range(0, input_bv.length(), 128):
110         if input_bv.length() < j+128:
111             # padding the last byte with 0s
112             bv = input_bv[j:] + BitVector(bitlist=[0] * (j+128-input_bv.length()))
113         else:
114             bv = input_bv[j:j+128]
115         # add round key
116         bv = bv ^ round_keys[0]
117         if j==0: print(bv.get_hex_string_from_bitvector())
118         # 13 round
119         for i in range(1,14):
120             # substitute bytes
121             bv = subBytes(bv)
122             if i==1 and j==0: print(bv.get_hex_string_from_bitvector())
123             bv = shiftRows(bv)
124             if i==1 and j==0: print(bv.get_hex_string_from_bitvector())
125             bv = mixColumns(bv)
126             if i==1 and j==0: print(bv.get_hex_string_from_bitvector())
127             # add round key
128             bv = bv ^ round_keys[i]
129             if i==1 and j==0: print(bv.get_hex_string_from_bitvector())
130         #last round
131         bv = subBytes(bv)
132         bv = shiftRows(bv)
133         bv = bv ^ round_keys[-1]
134         output_bv += bv
135     return output_bv # return the bit vector of the encrypted text for the whole
    content

```

```

134
135 def AES_Decrypt(fileName, round_keys):
136     FILEIN = open(fileName)
137     input_bv = BitVector(hexstring=FILEIN.read())
138     # create empty bit vector to store output
139     output_bv = BitVector(size=0)
140     # loop through all the input and extract 64 bit at a time
141     for j in range(0, input_bv.length(), 128):
142         if input_bv.length() < j+128:
143             # padding the last byte with 0s
144             bv = input_bv[j:] + BitVector(bitlist=[0] * (j+128-input_bv.length()))
145         else:
146             bv = input_bv[j:j+128]
147             # add round key
148             bv = bv ^ round_keys[0]
149             # 13 rounds
150             for i in range(1,14):
151                 bv = InvShiftRows(bv)
152                 bv = InvSubBytes(bv)
153                 bv = bv ^ round_keys[i]
154                 bv = InvMixColumns(bv)
155             #last round
156             bv = InvShiftRows(bv)
157             bv = InvSubBytes(bv)
158             bv = bv ^ round_keys[-1]
159             output_bv += bv
160     return output_bv # return the bit vector of the encrypted text for the whole
    content
161
162 def subBytes(bv):
163     c = BitVector(bitstring='01100011')
164     bv_out = BitVector(size=0)
165     for i in range(0, bv.length(), 8):
166         # extract 1 byte at a time,
167         # bv_out += subBytesTable[int(bv[i:i+4]) *10 + int(bv[i+4:i+8])]
168         a = bv[i:i+8].gf_MI(AES_modulus, 8) if int(bv[i:i+8]) != 0 else
        BitVector(intVal=0)
169         # For bit scrambling for the encryption SBox entries:
170         # scramble the bits of x' by XORing x' with
171         # four different circularly rotated versions of itself
172         # and with a special constant byte c = 0x63.
173         # The four circular rotations are through 4, 5, 6, and 7 bit positions to
        the right.
174         a1,a2,a3,a4 = [a.deep_copy() for x in range(4)]
175         a ^= (a1 >> 4) ^ (a2 >> 5) ^ (a3 >> 6) ^ (a4 >> 7) ^ c
176         bv_out += a
177     return bv_out
178 def shiftRows(bv):
179     #(i) not shifting the first row of the state array;
180     #(ii) circularly shifting the second row by one byte to the left;
181     #(iii) circularly shifting the third row by two bytes to the left;
182     #(iv) circularly shifting the last row by three bytes to the left.
183     #[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
184     # -> [0,5,10,15,4,9,14,3,8,13,2,7,12,1,6,11]
185     bv_out = BitVector(size=0)
186     for i in range(4):
187         a = 4*i
188         for j in range(4):
189             b = a + 5*j
190             if b <= 15:
191                 bv_out += bv[b*8:b*8+8]
192             else:
193                 bv_out += bv[(b-15-1)*8:(b-15-1)*8+8]
194     return bv_out
195 def mixColumns(bv):
196     # Each byte in a column is replaced by two times that byte,
197     # plus three times the next byte, plus the byte that comes next,
198     # plus the byte that follows.
199     bv_out = BitVector(size=0)
200     one = BitVector(intVal = 1, size = 8)
201     two = BitVector(intVal = 2, size = 8)
202     three = BitVector(intVal = 3, size = 8)

```

```

203 m = [[two,three,one,one],[one,two, three, one],[one, one, two, three], [three,
one, one, two]]
204 for i in range(4):
205     for j in range(4):
206         a = m[j][0].gf_multiply_modular(bv[8*i*4:8*i*4+8], AES_modulus, 8)
207         b = m[j][1].gf_multiply_modular(bv[8*(i*4+1):8*(i*4+1)+8], AES_modulus, 8)
208         c = m[j][2].gf_multiply_modular(bv[8*(i*4+2):8*(i*4+2)+8], AES_modulus, 8)
209         d = m[j][3].gf_multiply_modular(bv[8*(i*4+3):8*(i*4+3)+8], AES_modulus, 8)
210         bv_out += (a^b^c^d)
211     return bv_out
212 def InvShiftRows(bv):
213     # The first row is left unchanged,
214     # the second row is shifted to the right by one byte,
215     # the third row to the right by two bytes,
216     # and the last row to the right by three bytes
217     # [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
218     # -> [0,13,10,7, 4,1,14,11, 8,5,2,15, 12,9,6,3]
219     bv_out = BitVector(size=0)
220     for i in range(4):
221         a = 4*i
222         for j in range(4):
223             b = a - 3*j
224             if b >= 0:
225                 bv_out += bv[b*8:b*8+8]
226             else:
227                 bv_out += bv[(b+15+1)*8:(b+15+1)*8+8]
228     return bv_out
229 def InvSubBytes(bv):
230     d = BitVector(bitstring='00000101')
231     bv_out = BitVector(size=0)
232     for i in range(0, bv.length(), 8):
233         # bv_out += invSubBytesTable[int(bv[i:i+4])][int(bv[i+4:i+8])]
234         # For the decryption Sbox:
235         b = bv[i:i+8]
236         # For bit scrambling for the decryption SBox entries:
237         b1,b2,b3 = [b.deep_copy() for x in range(3)]
238         b = (b1 >> 2) ^ (b2 >> 5) ^ (b3 >> 7) ^ d
239         check = b.gf_MI(AES_modulus, 8)
240         b = check if isinstance(check, BitVector) else BitVector(intVal=0, size=8)
241         bv_out += b
242     return bv_out
243 def InvMixColumns(bv):
244     # Each byte in a column is replaced by two times that byte,
245     # plus three times the next byte, plus the byte that comes next,
246     # plus the byte that follows.
247     bv_out = BitVector(size=0)
248     oe = BitVector(hexstring = "0E")
249     ob = BitVector(hexstring = "0B")
250     od = BitVector(hexstring = "0D")
251     o9 = BitVector(hexstring = "09")
252     m = [[oe,ob,od,o9],[o9,oe,ob,od],[od,o9,oe,ob],[ob,od,o9,oe]]
253     for i in range(4):
254         for j in range(4):
255             a = m[j][0].gf_multiply_modular(bv[8*i*4:8*i*4+8], AES_modulus, 8)
256             b = m[j][1].gf_multiply_modular(bv[8*(i*4+1):8*(i*4+1)+8], AES_modulus, 8)
257             c = m[j][2].gf_multiply_modular(bv[8*(i*4+2):8*(i*4+2)+8], AES_modulus, 8)
258             d = m[j][3].gf_multiply_modular(bv[8*(i*4+3):8*(i*4+3)+8], AES_modulus, 8)
259             bv_out += (a^b^c^d)
260     return bv_out
261
262 if __name__ == "__main__":
263     genTables()
264     # read key from file, encrypt and expend is as 60 round keys (each 4 words)
265     round_keys = keyEncryptExpend()
266     # encrypt the message.txt with AES
267     # python AES.py -e message.txt key.txt encrypted.txt
268     # python AES.py -d encrypted.txt key.txt decrypted.txt
269     if sys.argv[1] == "-e":
270         # perform AES encryption on the plain text
271         encryptedText = AES_Encrypt(sys.argv[2], round_keys)
272         # transform the ciphertext into the hex string and write out to the file
273         with open(sys.argv[4], 'w') as f:

```

```
274         f.write(encryptedText.get_hex_string_from_bitvector())
275     # decrypt the message.txt with DES
276     elif sys.argv[1] == "-d":
277         # perform AES decryption on the encrypted.txt with round keys in the
            inversed order
278         decryptedText = AES_Decrypt(sys.argv[2], round_keys[::-1])
279         with open(sys.argv[4], "wb") as f:
280             decryptedText.write_to_file(f)
281
```