Implementation of AES

By Alexander Korobchuk

Overview:

This program is a basic implementation of 256-bit AES. It can take a key and plaintext and encrypt the plaintext using that key. The first part of the program performs key expansion on a key and constructs an output table like Table 6.3 from our textbook. It then performs AES on a specific plaintext using the same key from the previous part. It also constructs a table similar to Table 6.4 from the textbook. It finally repeats all of this using two self generated keys and plaintext.

Specifications:

Language: Python
Packages: None

Usage Walkthrough:

The program does not prompt for any input. The variables within the program must be changed directly in order to modify the output of the program.

Lines 6 and 7 are the key variable and the plaintext variable:

```
6   k = "1e0071c947d9e8591cb7add6af7f6798"  # This is our key.
7   plaintext = "0321456789abcdeffedcba9876543210"  # This is our plaintext.
```

These must be modified directly. Similarly, the custom key and custom plaintext are found at the bottom of the program, at lines 400 and 401:

```
400 custom_k = "6525cc8e81fa3701b59d255dd43f8938"  # This is our custom key.
401 custom_p = "989ce8c4ea33d4ade005b0fccb9578d6"  # This is our custom plaintext.
```

These are to be modified directly for the program to change its output.

The program begins by expanding the key and printing it. It then performs aes on the plaintext and key. After that, it will do the same thing with the custom key and custom plaintext:

```python
key_expansion(k)  # Expand the original key.
print("-------------------------------------------------------------------")
print()
perform_aes(plaintext, k)  # Perform aes on the given plaintext and key.

print()
print("-------------------------------------------------------------------")
print("BEGIN SELF GENERATED KEY AND PLAINTEXT HERE:")

custom_k = "6525cc8e81fa3701b59d255dd43f8938"  # This is our custom key.
custom_p = "989ce8c4ea33d4ade005b0fccb9578d6"  # This is our custom plaintext.

key_expansion(custom_k)  # Expand the custom key.
print("-------------------------------------------------------------------")
print()
perform_aes(custom_p, custom_k)  # Encrypt the custom plaintext with the custom key.
```

When running the program, it will output four tables. First it will output a table showing the key expansion of “1e0071c947d9e8591cb7add6af7f6798”. Here is part of the output:

```
Key Expansion using the key: 1e0071c947d9e8591cb7add6af7f6798

       Key Words                        Auxiliary Function
w0 = ['1e', '00', '71', 'c9']   RotWord = [['7f', '67', '98', 'af']]
w1 = ['47', 'd9', 'e8', '59']   SubWord = ['d2', '85', '46', '79']
w2 = ['1c', 'b7', 'ad', 'd6']   Rcon = ['01', '00', '00', '00']
w3 = ['af', '7f', '67', '98']   Z = ['d3', '85', '46', '79']

w4 = ['cd', '85', '37', 'b0']   RotWord = [['94', '15', 'a7', '39']]
w5 = ['8a', '5c', 'df', 'e9']   SubWord = ['22', '59', '5c', '12']
w6 = ['96', 'eb', '72', '3f']   Rcon = ['02', '00', '00', '00']
w7 = ['39', '94', '15', 'a7']   Z = ['20', '59', '5c', '12']

w8 = ['ed', 'dc', '6b', 'a2']   RotWord = [['ff', 'd3', 'd3', 'c8']]
w9 = ['67', '80', 'b4', '4b']   SubWord = ['16', '66', '66', 'e8']
w10 = ['f1', '6b', 'c6', '74']   Rcon = ['04', '00', '00', '00']
w11 = ['c8', 'ff', 'd3', 'd3']   Z = ['12', '66', '66', 'e8']

w12 = ['ff', 'ba', '0d', '4a']   RotWord = [['ae', 'ac', 'a6', 'a1']]
w13 = ['98', '3a', 'b9', '01']   SubWord = ['e4', '91', '24', '32']
w14 = ['69', '51', '7f', '75']   Rcon = ['08', '00', '00', '00']
w15 = ['a1', 'ae', 'ac', 'a6']   Z = ['ec', '91', '24', '32']

w16 = ['13', '2b', '29', '78']   RotWord = [['ee', '43', 'aa', '43']]
w17 = ['8b', '11', '90', '79']   SubWord = ['28', '1a', 'ac', '1a']
w18 = ['e2', '40', 'ef', '0c']   Rcon = ['10', '00', '00', '00']
w19 = ['43', 'ee', '43', 'aa']   Z = ['38', '1a', 'ac', '1a']
```

With the bottom of the table:

```
Final Expanded key: f28155f38f369ea2fc1167db3db182b
```

The second table performs AES on the plaintext “0321456789abcdeffedcba9876543210” using the same key as above. Part of the output:

```
Performing AES on plaintext: 0321456789abcdeffedcba9876543210
With the key: 1e0071c947d9e8591cb7add6af7f6798

Start of Round              After SubBytes              After ShiftRows        After MixColumns            Round Key
['03', '89', 'fe', '76']                                                                               ['1e', '47', '1c', 'af']
['21', 'ab', 'dc', '54']                                                                               ['00', 'd9', 'b7', '7f']
['45', 'cd', 'ba', '32']                                                                               ['71', 'e8', 'ad', '67']
['67', 'ef', '98', '10']                                                                               ['c9', '59', 'd6', '98']
['1d', 'ce', 'e2', 'd9'] ['a4', '8b', '98', '35'] ['a4', '8b', '98', '35'] ['a7', '94', '75', '66'] ('cd', '8a', '96', '39')
['21', '72', '6b', '2b'] ['fd', '40', '7f', 'f1'] ['40', '7f', 'f1', 'fd'] ['eb', '8e', '07', 'ba'] ('85', '5c', 'eb', '94')
['34', '25', '17', '55'] ['18', '3f', 'f0', 'fc'] ['f0', 'fc', '18', '3f'] ['48', '20', '8b', 'c7'] ('37', 'df', '72', '15')
['ae', 'b6', '4e', '88'] ['e4', '4e', '2f', 'c4'] ['c4', 'e4', '4e', '2f'] ['d4', 'd6', 'c6', 'c3'] ('b0', 'e9', '3f', 'a7')

['6a', '1e', 'e3', '5f'] ['02', '72', '11', 'cf'] ['02', '72', '11', 'cf'] ['1a', '5b', 'd6', 'b0'] ('ed', '67', 'f1', 'c8')
['6e', 'd2', 'ec', '2e'] ['9f', 'b5', 'ce', '31'] ['b5', 'ce', '31', '9f'] ['80', '72', '6b', '49'] ('dc', '80', '6b', 'ff')
['7f', 'ff', 'f9', 'd2'] ['d2', '16', '99', 'b5'] ['99', 'b5', 'd2', '16'] ['5b', '08', '00', 'cc'] ('6b', 'b4', 'c6', 'd3')
['64', '3f', 'f9', '64'] ['43', '75', '99', '43'] ['43', '43', '75', '99'] ['ac', '6b', '3a', 'ea'] ('a2', '4b', '74', 'd3')

['f7', '3c', '27', '78'] ['68', 'eb', 'cc', 'bc'] ['68', 'eb', 'cc', 'bc'] ['f6', '03', 'e2', 'f7'] ('ff', '98', '69', 'a1')
['5c', 'f2', '00', 'b6'] ['4a', '89', '63', '4e'] ['89', '63', '4e', '4a'] ['b4', 'dd', 'eb', 'a8'] ('ba', '3a', '51', 'ae')
['30', 'bc', 'c6', '1f'] ['04', '65', 'b4', 'c0'] ['b4', 'c0', '04', '65'] ['a4', 'f5', '48', '4d'] ('0d', 'b9', '7f', 'ac')
['0e', '20', '4e', '39'] ['ab', 'b7', '2f', '12'] ['12', 'ab', 'b7', '2f'] ['a1', 'c8', '70', 'ae'] ('4a', '01', '75', 'a6')

['09', '9b', '8b', '56'] ['01', '14', '3d', 'b1'] ['01', '14', '3d', 'b1'] ['0f', '3e', 'c5', 'dd'] ('13', '8b', 'e2', '43')
['0e', 'e7', 'ba', '06'] ['ab', '94', 'f4', '6f'] ['94', 'f4', '6f', 'ab'] ['b7', '1d', '50', 'ec'] ('2b', '11', '40', 'ee')
['a9', '4c', '37', 'e1'] ['d3', '29', '9a', 'f8'] ['9a', 'f8', 'd3', '29'] ['ea', '2b', '93', 'f5'] ('29', '90', 'ef', '43')
['eb', 'c9', '05', '08'] ['e9', 'dd', '6b', '30'] ['30', 'e9', 'dd', '6b'] ['6d', 'f9', '5a', '9c'] ('78', '79', '0c', 'aa')
```

With the bottom of the table:

```
Final Ciphertext: b4b0e04ab596a3f57cbf1c2f6d266bbb
```

The last two tables repeat the above processes. But this time it uses the custom key and custom plaintext. The outputs look exactly the same except with different characters.

Functions:

def init_key(key):

This function initializes the key in such a way that the key can be used by the program. We want a 4x4 matrix where each position in the matrix only contains 2 hex digits.

key: This is our inputted key.

<u>Pseudocode:</u>

Initialize a 2D matrix called *key_words* that is 4x4
Initialize *count* as 0
Initialize *count2* as 0

For *i* in range of the length of the key

If *i* mod 2 is 0
key_words[*count*][*count2*] = key[*i*]
key_words[*count*][*count2*] += key[*i* + *1*]
count2 += 1

If count2 >= 4
count2 = 0
count += 1
Return key_words

def rot_word(key_words):

This function performs a one-byte circular left shift on

key_words: This is our key

Pseudocode:

Initialize word variable as a 4x4 matrix with the rotated hex digits
For i in range of the length of word[0]
If the length of the word[0][1] = 1
Word[0][i] = 0 + word[0][i]

Return word

def sub_word(rotated_word, function):

This function substitutes the words. It uses the Rijndael S-Box to perform the substitutions.

rotated_word: This is our word after it's been rotated.
function: This tells the program if the rotated word is a 1D or 2D array. If 2D, function = 1. Otherwise its 0.

Pseudocode:

Initialized subbed_word list

If function = 0
For i in range of 4
Append the substituted sbox value to subword
Else if function = 1
Initialized subbed_word list as a 4x4 matrix
For i in range of 4
For j in range of 4

If the length of the rotated_word[j][i] = 1
Append a 0 to the front of rotated_word[j][i]
Append the substituted sbox value to subword
If the length of subbed_word[j][i] = 1
Append a 0 to the front of subbed_word[j][i]
Return the subbed_word

def xor_words(words1, words2):

This function XOR's 2 one dimensional word lists

words1: The first word list
words2: The second word list

Pseudocode:

Initialize xored_words list

Append words1[0] xored with words2[0] to xored_words
Append words1[1] xored with words2[1] to xored_words
Append words1[2] xored with words2[2] to xored_words
Append words1[3] xored with words2[3] to xored_words

Return xored_words

def xor_words_2d(words1, words2):

This function XOR's 2 two dimensional word lists

words1: The first word list
words2: The second word list

Pseudocode:

Initialize xored_words_2d as a 4x4 matrix

```
For i in range 4
    For j in range 4
        Append words1[j][i] XORed with words2[j][i] to xored_words_2d

For i in range of length of xored_words_2d[0]
    If the length of xored_words_2d[0][i] == 1
        Append a 0 to the front of xored_words_2d[0][i]
```

Return xored_words_2d

def print_table(key_words, rotated_word, subbed_word, rcon_position, z1, counter):

This function prints the table of the expanded key.

key_words: This is each stage of the key.
rotated_word: This is each stage of the rotated word
subbed_word: This is each stage of the substituted word
rcon_position: This is the position of the rcon table. We need it to print the right one.
z1: This is our stage of the z value.
counter: This keeps track of what position we are on.

Pseudocode:

```
For i in range 4
    For j in range 4
    If the length of the key_words[i][j] = 1
        Append a 0 to the front of key_words[i][j]

For i in range 4
    If the length of the rotated_word[0][i] = 1
        Append a 0 to the front of rotated_word[0][i]
    If the length of the subbed_word[i] = 1
        Append a 0 to the front of subbed_word[i]
    If the length of z1[i] = 1
        Append a 0 to the front of z1[i]

Print the counter, key_words[0], and rotated_word
Increase counter by 1
Print the counter, key_words[1], and subbed_word
Increase counter by 1
Print the counter, key_words[2], and rcon_position
Increase counter by 1
Print the counter, key_words[3], and z1
Increase counter by 1

Return the counter
```

def multiplication(a, b):

This function performs multiplication on the two inputted polynomials.

a: The first inputted polynomial.
b: The second inputted polynomial.

Pseudocode:

Initialize answer variable as 0

For *i* in range of 256
    answer XORed with a * ((b >> *i*) % 2) << i
    i increased by 1

Call reducer function with the answer

def reducer(answer, polynomial):

This is our reduction function. It takes an answer and will reduce it based on the irreducible polynomial. In this case, it will use the AES irreducible polynomial.

answer: This is the answer we wish to reduce.
polynomial: This is our irreducible polynomial we will use to reduce our answer.

Pseudocode:

Create a copy of the polynomial and call it *temp_poly*

While the length of the answer >= length of the polynomial
    temp_poly becomes a copy of the polynomial

    While the length of the answer > length of the *temp_poly*
        Append a ‘0’ to the *temp_poly*

    Initialize output list variable

    For *i* in range of the length of the answer
        If answer[*i*] is equal to *temp_poly[i]*
            Append a ‘0’ to output
        Else
            Append a ‘1’ to output

    Pop the first bit in output
    Answer becomes a copy of output
    Clear output

    While the first bit in answer = 0

Pop the first bit in answer

Initialize strings variable as a string of the answer
Join the strings
Put the answer into binary format
Return answer

def init_plaintext(plain_text):

This function will initialize the plaintext into a 2D array that we can use for our program. A simple string does not work so we need a 2D 4x4 matrix.

plain_text: This is our plaintext variable.

Pseudocode:

```
Initialize init_plain as a 4x4 matrix
Initialize count as 0
Initialize count2 as 0

For i in range of the length of plain_text
    If i mod 2 = 0
        Init_plain[count2][count] = plain_text[i]
        Init_plain[count2][count] += plain_text[i+1]
        count2 += 1

        If count2 >= 4
            count2 = 0
            count += 1
Return init_plain
```

def init_roundkey(key):

This function initializes the round key for encryption

key: This is our key variable

Pseudocode:

```
Initialize key_words 4x4 matrix
Initialize count as 0
Initialize coun2 as 0

For i in range of the length of the key
```

```
        If i mod 2 = 0
                key_words[count2][count] = key[i]
                key_words[count2][count] += key{i+1]
                count2  += 1
                If count2 >= 4
                        count2 = 0
                        count += 1
Return key_words
```

def shift_rows(subbed):

This function shifts the rows for encryption

subbed: This is the substituted key.

<u>Pseudocode:</u>

```
Initialize shifted as a 4x4 matrix
Shifted[0] = subbed[0]
shifted[0] = subbed[0]
shifted[1] = [subbed[1][1], subbed[1][2], subbed[1][3], subbed[1][0]]
shifted[2] = [subbed[2][2], subbed[2][3], subbed[2][0], subbed[2][1]]
shifted[3] = [subbed[3][3], subbed[3][0], subbed[3][1], subbed[3][2]]
return shifted
```

def mix_columns(shifted):

This function uses a circulant MDS matrix in order to “mix” the columns.

shifted: This is our shifted key

<u>Pseudocode:</u>

```
Initialize mixed as a 4x4 matrix
Initialize mixer as the circulant MDS matrix

For i in range of the length of mixer
        For j in range of the length of shifted[0]
                For z in range of the length of shifted
                        mixed [i][j] ^= multiplication of mixer[i][z] and shifted[z][j]
For i in range 4
        For j in range 4
                Mixed[i][j] is set to hexadecimal from binary
```

Return mixed

def print_table_enc(text, subbed, shifted, mixed, round_k):

This function prints the encrypted table

text: The first value to be printed
subbed: The subbed key
shifted: The shifted key
mixed: The mixed key
round_k: The round key

<u>Pseudocode:</u>

```
For i in range 4
        For j in range 4
                If length of text[i][j] = 1
                        Append a 0 to the front of text[i][j]
                If length of subbed[i][j] = 1
                        Append a 0 to the front of subbed[i][j]
                If length of mixed[i][j] = 1
                        Append a 0 to the front of mixed[i][j]
                If length of round_k[i][j] = 1
                        Append a 0 to the front of round_k[i][j]

For i in range 4
        Print text[i[], subbed[i], shifted[i], mixed[i], round_k[i]
```

def key_expansion(key):

This function performs our key expansion

key: The key to be expanded

<u>Pseudocode:</u>

```
Initialize i_key as init_key(key)

Initialize w_num as 0

For i in range 11
        If i = 0
                Rot = rot_word(i_key)
                Sub = sub_word(rot, 0)
```

```
        z  = xor_words(sub, rcon[0])
        W_num = print_table(i_kjey, rot, sub, rcon[0], z, w_num)
    Else if i = 10
        i_key[0] = xor_words(i_key[0], z)
        i_key[1] = xor_words(i_key[0], i_key[1])
        i_key[2] = xor_words(i_key[1], i_key[2])
        i_key[3] = xor_words(i_key[2], i_key[3])

        Print w_num, i_key[0]
        W_num += 1
        Print w_num, i_key[1]
        W_num += 1
        Print w_num, i_key[2]
        W_num += 1
        Print w_num, i_key[3]
        W_num += 1
Else
    i_key[0] = xor_words(i_key[0], z)
    i_key[1] = xor_words(i_key[0], i_key[1])
    i_key[2] = xor_words(i_key[1], i_key[2])
    i_key[3] = xor_words(i_key[2], i_key[3])
    Rot = rot_word(i_key)
    Sub = sub_word(rot, 0)
    z  = xor_words(sub, rcon[i])
    W_num = print_table(i_kjey, rot, sub, rcon[i], z, w_num)

For row in i_key
    Print row
```

def perform_aes(pt, key):

This function performs our AES encryption

pt: The plaintext to be encrypted
key: The key to be used in encryption

Pseudocode:

```
Initialize ct as init_plaintext(pt)
Initialize rk as init_roundkey(key)
Initialize kw as init_key(key)

For i in range 11
    If i = 0
```

```
        Rot = rot_word(i_key)
        Sub = sub_word(rot, 0)
        z  = xor_words(sub, rcon[0])
        W_num = print_table(i_kjey, rot, sub, rcon[0], z, w_num)
Else if i = 1
        kw[0] = xor_words(kw[0], z1)
        kw[1] = xor_words(kw[0], kw[1])
        kw[2] = xor_words(kw[1], kw[2])
        kw[3] = xor_words(kw[2], kw[3])
        rw = rot_word(kw)
        sw = sub_word(rw, 0)
        z1 = xor_words(sw, rcon[i])
        rk[0] = kw[0]
        rk[1] = kw[1]
        rk[2] = kw[2]
        rk[3] = kw[3]
        Rk = transpose of rk
        sub = sub_word(xored, 1)
        shifted = shift_rows(sub)
        mixed = mix_columns(shifted)
        print_table_enc(xored, sub, shifted, mixed, rk)
Else if i = 10
        kw[0] = xor_words(kw[0], z1)
        kw[1] = xor_words(kw[0], kw[1])
        kw[2] = xor_words(kw[1], kw[2])
        kw[3] = xor_words(kw[2], kw[3])
        rw = rot_word(kw)
        sw = sub_word(rw, 0)
        z1 = xor_words(sw, rcon[i])
        rk[0] = kw[0]
        rk[1] = kw[1]
        rk[2] = kw[2]
        rk[3] = kw[3]
        Rk = transpose of rk
        sub = sub_word(xored, 1)
        shifted = shift_rows(sub)
        mixed = mix_columns(shifted)
        print_table_enc(xored, sub, shifted, mixed, rk)
        Xored = xor_words_2d(shifted, rk)
        For i in range 4
                For j in range 4
                        If the length of xored[i][j] = 1
                                Append a 0 to the front of xored[i][j]
        Print xored[0]
```

```
		Print xored[1]
		Print xored[2]
		Print xored[3]
		Xored = transpose of xored
	Else
		kw[0] = xor_words(kw[0], z1)
		kw[1] = xor_words(kw[0], kw[1])
		kw[2] = xor_words(kw[1], kw[2])
		kw[3] = xor_words(kw[2], kw[3])
		rw = rot_word(kw)
		sw = sub_word(rw, 0)
		z1 = xor_words(sw, rcon[i])
		rk[0] = kw[0]
		rk[1] = kw[1]
		rk[2] = kw[2]
		rk[3] = kw[3]
		Rk = transpose of rk
		sub = sub_word(xored, 1)
		shifted = shift_rows(sub)
		mixed = mix_columns(shifted)
		print_table_enc(xored, sub, shifted, mixed, rk)

For i in range 4
	For j in range 4
		If length of xored[i][j] = 1
			Append a 0 to the front of xored[i][j]

Print xored
```

Testing:

The program was tested by checking with both the book as well as online resources. For example, testing the encryption and key expansion was done by inputting the example in the book in order to see if the program outputted the same thing. Then, the given homework key and plaintext were checked with an online AES encryption tool, making sure that the output was the same from my program and the online tool.