The Avalanche Effect

By Alexander Korobchuk

Overview:

This program shows the avalanche effect using one key and 4 different plaintexts that all differ by only one bit. The program prints tables of the avalanche effect and shows how many bits are different in the end.

The Avalanche effect is a desirable property in cryptographic algorithms because it is if one bit is changed, the output is changed drastically. It is desirable to have at least half of the output bits to be different if only one input bit is changed.

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.

Line 440 is where the key is:

```
440    key_1 = "0f0071c947d9e8591cb7add6af7f6798"
```

Line 441, 443, 450, and 457 hold the different plaintexts that will be tested:

```
441    plaintext_1 = "0200456789abcdeffedcba9876543210"
```

```
443    plaintext_2 = "0200456789abcdeffedcba9876543211"
```

```
450    plaintext_2 = "0200456789abcdefeedcba9876543210"
```

```
457    plaintext_2 = "0200446789abcdeffedcba9876543210"
```

The program begins by performing AES on the first plaintext with the key. It then performs AES on the second plaintext with the first key. It then prints the avalanche table using the results, and then it clears the avalanche_list which holds the various plaintexts in the different stages:

```
print("Avalanche Effect #1")
key_1 = "0f0071c947d9e8591cb7add6af7f6798"  # This is our key.
plaintext_1 = "0200456789abcdeffedcba9876543210"  # This is our plaintext.
perform_aes(plaintext_1, key_1)
plaintext_2 = "0200456789abcdeffedcba9876543211"
perform_aes(plaintext_2, key_1)
print_av_table(avalanche_list)
avalanche_list.clear()
```

It repeats the above process two more times, but using the different one bit flipped plaintexts.

An example of the output:

```
Avalanche Effect #1
Round                                                                    Number of Bits that Differ
                        0200456789abcdeffedcba9876543210                              1
                        0200456789abcdeffedcba9876543211

0                       0d0034aece7225b6e26b174ed92b5588                              1
                        0d0034aece7225b6e26b174ed92b5589

1                       9d1d0cf10fd2ff3ff2ecf9f9f7094cfa                              16
                        fe7ea9370fd2ff3ff2ecf9f9f7094cfa

2                       306ede104aaba649618c36c7606783d6                              75
                        e18b3b247190eb3f4cfb6cea99c9d481

3                       b383e0f0b4c10d671eb3e29742f17ae1                              66
                        f97ea157bb695fd4d0914bfd4f9da44e

4                       301d84e26b5f50679a36f1cb6b0bb5ee                              57
                        a75ac5face8284049a72fd60c5694da9

5                       7280ef768ea9ed611c0a43a1d648e5f5                              56
                        b3848c5a3138f464ffd2e164927e7416

6                       92b2e946c8387566878a51632d136ed4                              67
                        ecb0375ecf027e916723f8bf72c1199e

7                       1d35f3ae2d076dcc47282312a68c5133                              65
                        47bcfc50480ac77d41956a417beecbc3

8                       8e6f6c930f3c4de5bfe00c8a20370cc6                              63
                        b2b5aed85b749cc52433c56d9490cb54

9                       face6abc9fdd98774b6ddaa450765e23                              53
                        e16e2adc3b53f004e904ec56991c0a43

10                       cb045eee544ec6fc5b2904b3e0505e59                              70
                        1e28c7d26d80f91ff68de3b45d4494a1
```

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.

Pseudocode:

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

<u>Pseudocode:</u>

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

<u>Pseudocode:</u>

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 multiplication(a, b):

This function performs multiplication on the two inputted polynomials.

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

<u>Pseudocode:</u>

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.

<u>Pseudocode:</u>

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.

<u>Pseudocode:</u>

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

<u>Pseudocode:</u>

```
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

Pseudocode:

```
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 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)
        Append the transpose of xored to the avalanche_list
        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
        Append xored to the avalanche list
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]

Append the transpose of xored to avalanche_list

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

def differing_num(pt_1, pt_2):

This function calculates the number of different bits from the plaintext and returns it.

pt_1: The first plaintext
pt_2: The second plaintext

<u>Pseudocode:</u>

```
If pt_1 is not a list
        Hex1 becomes pt_1
        Hex2 becomes pt_2
Else
        For i in range 4
                Append the strings of pt_1[i] to hex1
        For i in range 4
                Append the strings of pt_2[i] to hex2

Initialize count as 0

For i in range 256
        If hex1 >> i & 1 != hex2 >> i & 1
```

```
        Count = count + 1
Return count
```

def print_av_table(list_of_text):

This function prints the avalanche table

list_of_text: This contains all of the keys from the various stages

<u>Pseudocode:</u>

```
Avalanche_list pop 0
Avalanche_list pop 11
Avalanche_list pop 12
Avalanche_list pop 22

Print plaintext_1
Initialize diff as differing_num(plaintext_1, plaintext_2)
Print diff
Print plaintext_2

Initialize counter as 0
Initialize first_t as 0
Initialize second_t as 11

For i in range 11
    Print counter
    For row in avalanche_list[first_t]
        Print row
    Diff = differeng_num(avalanche_list[first_t], avalanche_list[second_t]
    Print diff
    For row in avalanche_list[second_t]
        Print row
    Counter += 1
    First_t += 1
    second_t += 1
```

Testing:

The program was tested by comparing results to the book. Once the program could successfully replicate the table from the book, then it was assumed that the other avalanche tables were accurate as well. The other plaintexts were checked by also comparing the number of bits different, and making sure they were correct.