Cryptography report

Katarzyna Badio

20 czerwca 2023

1 The goal of the project

The objective of this project is to create one hundred random s-boxes with 512 values, because for each 256 valued s-box, an inverse s-box is calculated. Then for each s-box the goal was to calculate nonlinearity, sac property (strict avalanche criteria) and xor profile. After this, the results are presented in a graphical form.

2 S-box creation

S-boxes are used in symmetric AES algorithm, which is a block cipher. First, random 256 values are created in a function *generate arr*, which is seen in a Listing 1. These values are stored as binary numbers, because these are used to create a set of eight functions (Listing 3). For example, each first bit of every 256 numbers in an s-box is appended to the function one in *set of functions*. So, the set is of length two hundred and fifty-six. The same applies to the second, third and until eigth bit.

```
def generate_arr():
    arr = []
    for integer in range(256):
        integer = random.randint(0, 256)
        arr.append(bin(integer))
    return arr
```

Listing 1: Random generator

```
def  prepare_set_of_functions(_arr):
    set_of_functions = [[] for _ in range(8)]

    for el in _arr:
        el_str = el[2:]

        if len(el_str) != 8:
            diff = 8 - len(el_str)
            zeros_front = '0'*diff
            el_str = zeros_front + el_str

        for i in range(8):
            set_of_functions[i].append(el_str[i])

    return set_of_functions
```

Listing 2: Prepare functions

3 Nonlinearity

Nonlinearity is a property which ensures that the relationship between the input and the output values is highly complex and not easily predictable. The lower the output value is, the better is the security, because the value provides a measure how far the S-box deviates from a linear mapping.
Truth table is created (line 6 of Listing 1). Then, all combinations of x0, x1, ... x7 labels are prepared (line 8 of Listing 1), which is done by a function *generate sets*. The labels are for naming columns of a truth table. For example, x0 contains repeating pattern of '01' and x1 has a pattern of '0011'.

Nonlinearity is measured by calculating a hamming distance (D) and then calculating weight (f). In line 22 of Listing 1, in *find minimum sum xor fx*, every function from previously prepared set of eight functions (Listing 1) is xored with results, stored in *xor sets*. Then the sum of it is calculated (f). The minimum sum is returned as the result.

$$D = f(x) \oplus g'(x) \tag{1}$$

$$f = w(f(x) \oplus g'(x)) \tag{2}$$

```
1  def calculate_hamming_distance(_set_of_functions):
2
3      labels = ['x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7']
4
5      # create labels for generatedtruth table and create sets
       of 2, 3...
6
7      truth_table = helper.generate_truth_table(256)
8      functions = dict(zip(labels, truth_table))
9      combinations= generate_sets(labels)
10
11
12     # for eachcombination, return a xor result
13     xor_sets = []
14     for i in range(len(combinations)):
15         for comb in combinations[i]:
16             if comb not in xor_sets:
17                 partial_res = generate_xor_result_for_x(comb,
       functions)
18                 xor_sets.append(partial_res)
19                 xor_sets.append(generate_inverse(partial_res)
       )
20
21
```

```
22      # find result betweeneach function and xor sets
23      mini_all = 250
24      for i, funcin enumerate(_set_of_functions):
25          mini = find_minimum_sum_xor_fx(change_to_nums(func),
    xor_sets)
26          # print("Minimumsumfor function " + str(i)  + " is "
     + str(mini))
27          if mini < mini_all:
28              mini_all = mini
29
30      return mini_all
```

Listing 3: Hamming distance

4 Xorprofile

An XOR profile of an S-box involves examining the inputs and outputs of the S-box and counting the number of input-output pairs that differ by a single bit (i.e., differ in exactly one position). This process is done for all possible input pairs.
To calculate xor profile, first the truth table is generated with 256 values (line 2 in Listing 4). Then, from a truth table, rows are created, in a function *create x rows* (Listing 5). Then, in a function in a Listing 6, in line 5 and line 6, two neighboring x values are chosen in range of length *x rows*. For every two values, the xor value is calculated and it will be a row index of the table. Also, these two values are indexes of values from an S-box table and these are retrieved, swapped to binary numbers and xored (line 22).
The result of xor is a column index of the table. So, in the resulting array (*result array*), a value is indexed and 2 is added to a sum at that position (line 27). The resulting xor profile is the maximum value from that table (Listing 4).

A high xor profile indicates good non-linearity and cryptographic properties of the S-box.

```
1 def calculate_xor_profile(arr):
2     truth_table = helper.generate_truth_table(256)
3     rows = xor_profile.create_x_rows(truth_table)
4     _, result_array = xor_profile.create_quadruples(arr,rows)
5     maxi, _ = xor_profile.find_max(result_array)
6     return maxi
```

Listing 4: Calculate xor profile

```
1 def create_x_rows(_truth_table):
2     x_rows = []
```

```
    for col in range(len(_truth_table[0])):
        temp = []
        for row in range(len(_truth_table)-1, -1, -1):
            temp.append(_truth_table[row][col])
        x_rows.append(temp)
    return x_rows
```

Listing 5: Create x rows

```
def create_quadruples(arr, _x_rows):
    quadruples = []
    result_array = [[0 for _ in range(256)] for _ in range
(256)]

    for i in range(len(_x_rows)):
        for j in range(i+1, len(_x_rows)):

            xor_x = []
            for xor_x in zip(_x_rows[i], _x_rows[j]):
                xor_x.append(v1 ^ v2)

            y1 = [int(v) for v in str(find_value_in_sbx(arr,
_x_rows[i]))[2:]]
            while len(y1) != 8:
                y1.insert(0, 0)
            y2 = [int(v) for v in str(find_value_in_sbx(arr,
_x_rows[j]))[2:]]
            while len(y2) != 8:
                y2.insert(0, 0)
            xor_y = []
            for xor_y in zip(y1, y2):
                xor_y.append(v1 ^ v2)

            xor_x = bytearr_to_int(xor_x)
            xor_y = bytearr_to_int(xor_y)
            result_array[xor_x][xor_y] += 2
            quadruples.append((_x_rows[i], _x_rows[j], xor_x,
xor_y))

    return quadruples, result_array
```

Listing 6: Create quadruples

5 Sac property

The SAC property refers to the behavior of an S-box where a single input bit flip should ideally cause approximately half of the output bits to flip on average. In other words, if a single input bit is changed, the resulting output bits should change with equal probability, achieving a balanced and unpredictable transformation.
For each function from *set of functions*, result of xor with lambda is cal- culated in line 8. The details of a function are in Listing 8. For example for lambda one, first value from a function is stored as a second one, and a second one as a first one, so these are swapped until all values of a function are swapped correctly. For lambda two, first two values are stored as a third and fourth one, and these are stored as a first and second. It is easy to see a pattern of swapping numbers there.

Then, the results of swap, are xorred with the function from *set of functions* (line 9 in Listing 7). After this, the average percentage of flipped bits is re- turned.

```
1  def calculate_sac(arr,  set_of_functions):
2      min_p_avg= 200
3      for f in range(len(set_of_functions)):
4          l_iters = [1,  2,  4,  8,  16,  32,  64,  128]
5          p_sum  0
6          for iter  in l_iters:
7              result_f_lambda = sac.calculate_f_lambda(
       set_of_functions[f],  iter)
8              xor = sac.calculate_xor(set_of_functions[f],
       result_f_lambda)
9              p_sum+= sac.count_percentage(xor)
10         p_avg = p_sum/8
11         if  p_avg < min_p_avg:
12             min_p_avg= p_avg
13     return min_p_avg
14 
```

Listing 7: Calculate SAC

```
1  def calculate_f_lambda(_function, _z):
2  
3      result_f = []
4      switch = 'a'   # append
5      counter =0
6      for i in range(0, len(_function),  _z):
7          while counter < _z:
8              if switch  == 'a':
9                  result_f.append(_function[i+counter])
10             else:
```

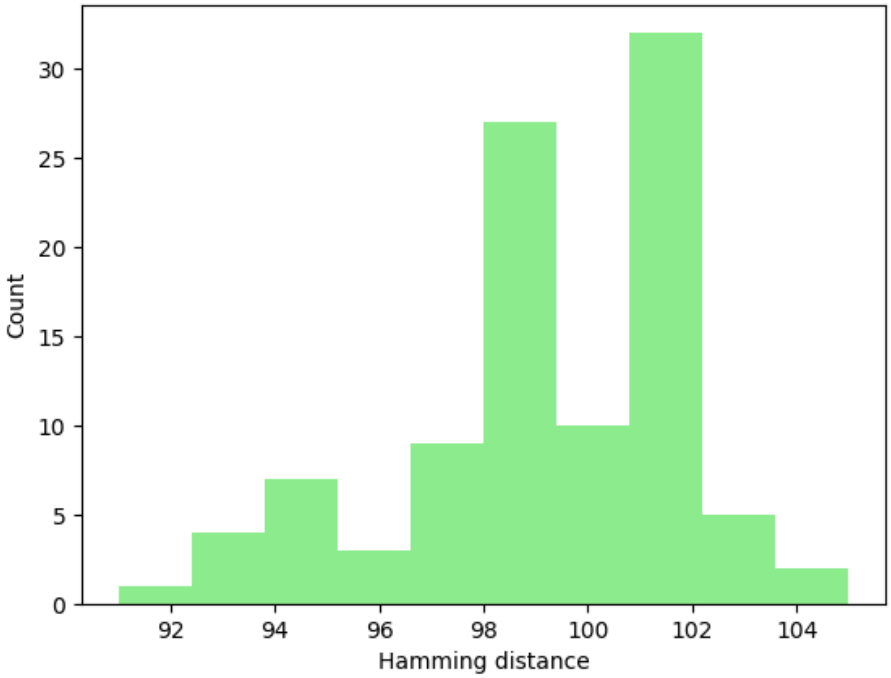
```
                result_f.insert(i-_z+counter,    _function[i+
    counter])
            counter += 1

        # after  a round, switch from appendto  prepend...
        if  counter == _z:
            counter = 0
            if  switch == 'a':
                switch=  'p'
            else:
                switch = 'a'

    return result_f
```

Listing 8: Calculate f lambda

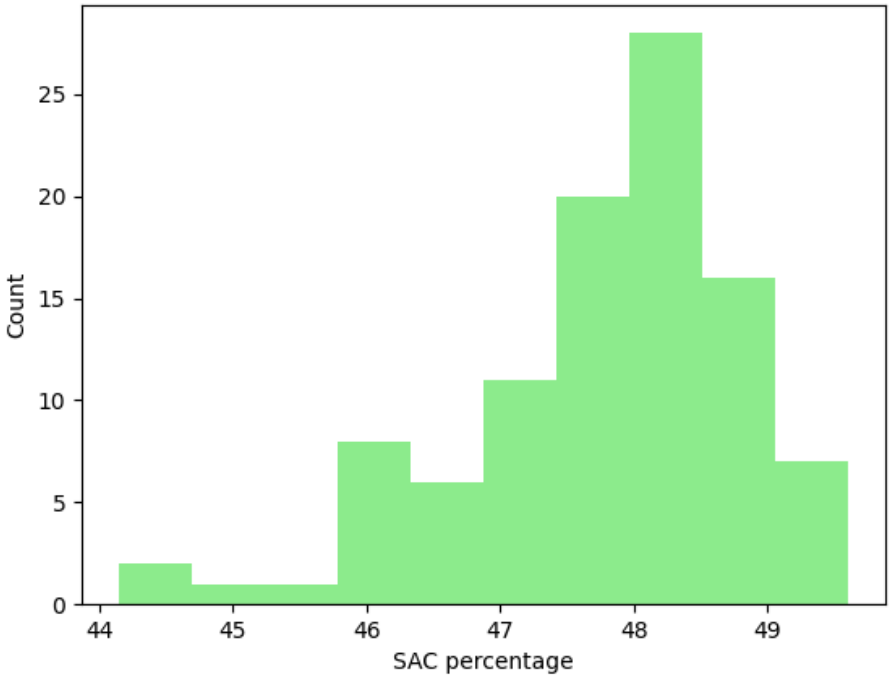
6 Results

In the picture number 1, some results of calculating Hamming distance are presented. The best nonlinearity is between 91 and 92. Most values oscillated around 99 and 101.

The results of calculating SAC is seen in the picture 2. The peak is around value 48.



Rysunek 1: Hamming distance



Rysunek 2: SAC property