Particle Swarm Optimization based on S-Boxes Generation

Alexandr Kuznetsov[1,2], Yaroslav Derevianko[1,2], Nikolay Poluyanenko[1,2], and Oleksandr Bagmut[1]

[1] *V. N. Karazin Kharkiv National University, 4 Svobody sq., Kharkiv, 61022, Ukraine*
[2] *JSC "Institute of Information Technologies," 12 Bakulin str., Kharkiv, 61166, Ukraine*

Abstract

The generation of nonlinear substitutions (S-boxes) is an important task in the design of modern symmetric cryptoalgorithms. Various cryptographic properties of S-boxes (nonlinearity, balance, delta-uniformity, correlation and algebraic immunity, etc.) characterize their resistance to linear, differential, algebraic and other cryptanalysis methods. This article explores a computational particle swarm optimization (PSO) method as applied to the problem of generating nonlinear substitutions. Having a set of possible solutions (particles) and moving these particles in the search space, the PSO tries to improve the possible solution in terms of some quality indicator. We use nonlinearity, balance, delta uniformity, algebraic immunity and linear redundancy as the main indicators, and randomly generated S-boxes are used as a set of particles. This article shows several PSO modifications for generating nonlinear substitutions. At first, we reproduce the previously known PSO modification for generating S-boxes and show its low efficiency. At second, we propose our own PSO implementation and show that this method can actually generate substitutions with high cryptographic properties. The experimental results allow us to establish the influence of the size of the population of particles and the number of iterations of the outer loop on the efficiency of the heuristic generation of nonlinear substitutions. In addition, we explore the similarity of the generated substitution tables with the AES cipher S-box.

Keywords

Nonlinear substitutions, s-boxes, particle swarm optimization, computational search.

1. Introduction

The efficiency of symmetric crypto algorithms is determined by many factors [1–3]: basic transformation structure; key schedule scheme; properties of the strength of crypto primitives, etc.

One of the basic primitive blocks of symmetric cryptography is nonlinear substitutions (S-boxes, substitution tables) [4–6]. Nonlinearity, balance, delta-uniformity, correlation and algebraic immunity and other properties characterize resistance of S-boxes to linear, differential, algebraic and other cryptanalysis methods. Therefore, the problem of generating substitutions with the required properties is an important and urgent scientific problem [7–9].

Various methods are used to generate cryptographically strong substitutions, see for example [6,10,11]. In one of the latest works [12], it was proposed to use the particle swarm method (PSO). However, the results shown in this article are not reliable (see for example our comment [13]). However, the idea of using PSO to generate S-boxes can be useful. The purposes of our article are researching of the PSO for heuristic generation of nonlinear substitutions, experimental verification of the results, and justification of some PSO parameters to form S-boxes with the required properties.

CPITS-II-2021: Cybersecurity Providing in Information and Telecommunication Systems, October 26, 2021, Kyiv, Ukraine
EMAIL: kuznetsov@karazin.ua (A. Kuznetsov); yarik0009258@gmail.com (Y. Derevianko); nlfsr01@gmail.com (N. Poluyanenko); oleksandr.bagmut@karazin.ua (O. Bagmut)
ORCID: 0000-0003-2331-6326 (A. Kuznetsov); 0000-0002-3290-3373 (Y. Derevianko); 0000-0001-9386-2547 (N. Poluyanenko); 0000-0003-3241-5756 (O. Bagmut)

2. Particle Swarm Optimization

Particle swarm optimization - heuristic method of global optimization, which is implemented on the basis of populations, which is proposed in [14]. PSO is based on "swarm" intelligence, which is a natural behavior of birds, fishes, insects and others. Swarm particles either fly away or fly together in search of food before they find a good place, where the food is located. While searching for food, there is always one bird that feels and seeks food better than others, so such bird is more likely to find a place where food can be found, which means that it will have better information about the source of food than others. Since birds constantly share such "good" information in the search process, the flock will eventually move to a "better" place where food is more likely to be found. In PSO, the movement of "birds" begins from one location to another, equivalent to a flock, good information is equivalent to the most optimistic solution, and the expected food is equivalent to the most optimistic solution for the entire operation of the algorithm.

This method helps to find the most optimistic solution due to the cooperation of each element of the population, the “bird.” Quite complex optimization problems can be solved using PSO algorithm. The advantages allow it to be applied to many areas of optimization alone and in combination with other existing algorithms. The algorithm is used in such areas as neural network training, optimization of certain functions, machine learning, signal processing, etc., [15–18].

In the basic PSO algorithm, the population consists of «N» particles, and the location of each of the particles corresponds to a potential solution in d -dimensional space. The position of each particle in the swarm is influenced by both the most optimistic position during its movement (individual experience, called personal best - $pBest$ of the particle), and the position of the most optimal particle in its vicinity (experience that is called the best of all - $gBest$).

Flock particles fly into search space due to their exploration and exploitation capabilities and use $pBest$ and $gBest$, to find the best solution in PSO. In addition, each particle is characterized by velocity. The velocity and position of each particle are reviewed after each subsequent iteration of the algorithm.

The velocity and position of each particle are determined at each step as:

$$v_{id}^{k+1} = v_{id}^{k} + c_1 r_1 (pBest_{id}^{k} - x_{id}^{k}) + c_2 r_2 (gBest_{id}^{k} - x_{id}^{k}) \quad (1)$$

$$x_{id}^{k+1} = x_{id}^{k} + v_{id}^{k+1} \quad (2)$$

where:

- v_{id}^{k} and x_{id}^{k} is velocity and position of the particle «i» at its «k» times and the d-dimension quantity of its position;
- $pBest_{id}^{k}$ is d -dimension quantity of the individual «i» element at its most optimist position;
- $gBest_{id}^{k}$ is d -dimension quantity of the most optimistic position of the whole «swarm»;
- parameters c_1, c_2, r_1, r_2 are randomly generated within $[0,1]$.

In abstract [12] proposed interpretation of PSO for the generation of highly nonlinear S-boxes that implement bijective mappings $S : \{0,1\}^8 \to \{0,1\}^8$. Although, the authors of this research made mistakes in calculating the nonlinearity of S-boxes (in particular, see our commentary [13]), we reproduced their proposed algorithm for computational search of nonlinear substitutions and made our own researches of the efficiency of PSO.

2.1. PSO Implementation to Generate S-Boxes

According to the description from [12] the generation of S-boxes occurs using the following algorithm.

1. Population initialization

For S-box optimization problem, each individual 8×8 S-box is considered as the particle. The S-box population is generated randomly so that the S-Boxes remain bijective. Generation is repeated until the N -sized population is filled.

2. Calculation of nonlinearity

Next, nonlinearity is calculated for each block. According to this nonlinearity, the population of blocks is sorted (in descending order of nonlinearity).

3. PSO vector initialization

The velocity vector is filled with zeros and updated at each successful iteration. Each position vector is initialized using the appropriate S-Box in the population. The velocity vector is updated by formula (1), and the location vector is updated by formula (2). The vector of the most optimistic positions during the iteration ($pBest$) is updated for each generated population if the nonlinearity values of the new blocks are better than the previous ones. The most optimal of all is the particle with the highest nonlinearity in the population.

4. Initialization of PSO parameters

PSO parameters, such as c_1, c_2, r_1 and r_2 are randomly selected using a Renyi map. During the optimization phase of the algorithm, these parameters randomly change at each iteration. Another parameter is the coefficient of inertia (inertial weight), which is given by the formula:

$$w_{curIter} = w_1 + (curIter - 1)(\frac{w_2 - w_1}{maxIter}) \quad (3)$$

where w_1 and w_2 is the initial and final value of the coefficient, respectively.

5. Improvement and adjustment

The velocity and location vectors are updated according to the laws in formulas (1) and (2) respectively. The process of such improvement generates certain values that are repeated or negative. To prevent this, the authors use certain processing methods, replacing repeated values with values that are lacking to preserve the bijectivity. An algorithm for such improvements is not provided in [12].

6. The final step of the iteration

Next, the nonlinearity value is also calculated for all newly generated blocks, and all S-Boxes, including those already in the population, are sorted again in descending of nonlinearity. N best blocks according to the nonlinearity remain in the population, all other blocks are discarding. Vectors $pBest$ and $gBest$ are updated as above.

The pseudocode of the algorithm presented in the article [12], is shown in Fig. 1.

In [13] we showed that the calculation of the nonlinearity of S-Boxes in [12] was incorrect. According to Fig. 1 nonlinearity is chosen as the main target of optimization. Therefore, the effectiveness of the use of PSO to generate substitutions can't be established by publication [12].

In this article, we reproduced the algorithm shown in Fig. 1 and made some experimental researches with correct calculation of nonlinearity. Unfortunately, we were unable to form a single S-Box with a nonlinearity greater than 98, even after numerous experiments. This is a very poor result, which indicates an unsuccessful PSO interpretation. But this does not mean that the PSO method cannot be efficiency applied in another interpretation.

2.2. New PSO Implementation to Generate S-Boxes

In this work, we propose a new implementation of PSO. The PSO method modified by us almost completely repeats algorithm from [12]. The main difference is in the first iteration of the while loop when forming the first block in the new population, as well as when filling the vectors of the most optimal blocks. Additionally, we estimate other properties of S-Boxes (algebraic immunity, delta uniformity, and linear redundancy).

The essence of the modified algorithm is given below.

1. Population initialization

As in the algorithm discussed earlier, each 8×8 S-box is considered as the particle. The S-box population is generated randomly so that the S-Boxes remain bijective. Generation is repeated until the N -sized population is filled.

2. Calculation of nonlinearity

Next, nonlinearity is calculated for each block. According to this nonlinearity, the population of blocks is sorted (in descending order of nonlinearity).

```
_____________________________________________
Input arguments:
N ← number of blocks in the population
max_itr ← maximum number of iterations
xr ← initial value of Renyi chaotic map
c ← parameter of Renyi map
_____________________________________________
Generation of the initial population of S-boxes:
xr ← renyi_map(xr, c, 100)
population ← zeros(2 × N, 256) // 256 for 8 × 8 S-boxes
for i ← 1 to N
     [sbox_i, xr] ← gen_sbox(xr, c)
     population[i] ← sbox_i
end for
population[1] ← aes_sbox
_____________________________________________
Calculation of nonlinearity for each of the particles:
for i ← 1 to N do:
NL[i] ← nonlinearity(population[i])
end for
NL_sorted ← sort(NL) // sorting in descending order
gBest ← population[1]
pBest_i ← population[i]
Vel ← zeros(N, 256)
Setting the inertial weight w
_____________________________________________
The beginning of the improvement phase:
While (max_itr > 0) do:
     xr ← renyi_map(xr, c); c1 ← 2*xr
     xr ← renyi_map(xr, c); c2 ← 2*xr
     xr ← renyi_map(xr, c); r1 ← xr
     xr ← renyi_map(xr, c); r2 ← xr
     NL ← NL_sorted
     for i ← 1 to N do:
              for j = 1 to 256 do:
                        Vel[i][j]  ←  ceil(w*Vel[i][j]  +  c1*r1*(pBest[i][j]  -  population[i][j])  +
                     c2*r2*(gBest[j]-population[i][j]))
                     if (Vel[i][j] < 0)
                              Vel[i][j] ← (Vel[i][j] + 256)mod(256)
                     end if
                     X[i, j] ← int(population[i][j] + Vel[i][j])mod(256)
                     temp_sbox[j] ← X[i, j]
              end for
                 Apply a specific control algorithm to preserve the bijecivity int temp_sbox
              population[N + i] ← temp_sbox
     end for
     for i ← 1 to N do:
              NL_sorted[N + i] ← nonlinearity(population[N + i])
     end for
     NL_sorted ← sort(NL_sorted)
     Discarding of redundant elements
     for i ← 1 to N do:
              if (NL[i] < NL_sorted[i])
                        pBest[i] ← population[i]
              end if
     end for
     Update gBest
     max_itr ← max_itr - 1
end while
_____________________________________________
```

Figure 1: Implementation of PSO from the article [12]

3. PSO vector initialization

The velocity vector is filled with zeros and updated at each successful iteration. Each position vector is initialized using the appropriate S-Box in the population. The velocity vector is updated by formula (1), and the location vector is updated by formula (2). The difference in our algorithm is that when forming a new population, the first block of the new population is formed due to the interaction of the first block of the initial population with itself and changing a small number of values. This gives an almost zero velocity vector on the first iteration, which allows to gradually deteriorate on subsequent iterations. In subsequent iterations, if after applying this method the nonlinearity of S-box is higher than 98, the block is further mixed randomly to improve other parameters such as linear redundancy, delta uniformity and algebraic immunity.

This is done in order to reach a compromise, so that there are no situations where one parameter is very good and the others are bad. It is possible to ensure that all parameters are sufficient and satisfy most conditions using this method.

The vector of the most optimistic positions during the iteration (*pBest*) is updated for each generated population if the nonlinearity values of the new blocks are better than the previous ones.

The most optimal particle in our method is also updated at each iteration and equated to the best block in the vector of the most optimistic positions *pBest* .

4. Initialization of PSO parameters

PSO parameters such as c_1, c_2, r_1 and r_2 are randomly selected. During the optimization phase of the algorithm, these parameters randomly change at each iteration. Inertial weight is also (3).

5. Improvement and adjustment

The following algorithm was developed to preserve the objectivity of S-Boxes (Fig. 2). After forming a block, each of its elements is checked for coincidence with another element in this block. If when checking a certain element such an element already exists in this block, then the variable *contains* becomes 1, and the value is updated by adding a random value to it, and taking it by modulo 256. This is repeated as long as only unique values from 0 to 255 remain in the S-Box. The pseudocode of the algorithm is given below.

```
for (int i = 0; i < N; ++i)
     for (int j = 0; j < size;)
              Updating values by (1) and (2)
            int contains;
            if (contains == 0)
                tempSbox[j] = X;
            end if
              else
                tempSbox[j] = myModulusDec((tempSbox[j] + rand()), 256);
            end else;
            contains = 0;
            for (int k = 0; k < j; ++k)
                if (tempSbox[k] == tempSbox[j])
                    contains = 1;
                    break;
                end if
            end for
            if (!contains)
                j++;
              end if
     end for
     ......
end for
```

Figure 2: Algorithm for preserving the bijectivity of population particles

6. The final step of the iteration

Next, the nonlinearity value is also calculated for all newly generated blocks, and all S-Boxes, including those already in the population, are sorted again in descending of nonlinearity. N best blocks according to the nonlinearity remain in the population, all other blocks are discarding. Vectors *pBest* and *gBest* are updated as above.

The pseudocode of the modified algorithm is shown in Fig. 3.

Our proposed new modification of PSO allows to form S-boxes with nonlinearity 104, delta uniformity 8, linear redundancy 0 and algebraic immunity 3 in a relatively short time. As an example, we give one of the following S-boxes (8-bit output vectors are given in decimal format):

```
{99, 124, 119, 123, 242, 107, 111, 197, 48, 1, 103, 43, 254, 215, 171, 180,
  202, 130, 201, 125, 250, 89, 71, 240, 173, 212, 162, 175, 156, 164, 114,
 192, 183, 253, 147, 38, 54, 63, 29, 146, 52, 165, 229, 241, 113, 216, 49,
 88, 4, 199, 35, 195, 24, 150, 5, 154, 7, 18, 128, 226, 235, 39, 178, 117,
 9, 131, 44, 26, 27, 110, 90, 160, 82, 59, 214, 179, 41, 227, 47, 132, 83,
  209, 0, 237, 32, 252, 177, 91, 106, 203, 190, 57, 74, 76, 51, 207, 208,
  239, 170, 251, 67, 77, 166, 133, 69, 249, 2, 127, 80, 60, 159, 168, 81,
 163, 64, 143, 221, 28, 56, 245, 188, 182, 218, 33, 16, 255, 243, 210, 205,
12, 19, 236, 95, 151, 68, 23, 196, 167, 126, 61, 100, 93, 25, 115, 96, 129,
79, 220, 34, 42, 144, 136, 70, 238, 184, 20, 222, 94, 11, 219, 224, 50, 58,
  10, 73, 6, 36, 92, 194, 211, 172, 98, 145, 149, 228, 121, 231, 200, 55,
 109, 141, 213, 78, 169, 108, 86, 244, 234, 101, 122, 174, 8, 186, 120, 37,
 46, 187, 232, 72, 198, 85, 247, 116, 31, 191, 189, 139, 138, 112, 62, 181,
  102, 204, 3, 75, 17, 97, 53, 87, 185, 134, 193, 148, 158, 225, 248, 152,
223, 105, 217, 142, 176, 155, 30, 135, 233, 206, 40, 45, 22, 140, 161, 137,
         13, 153, 230, 66, 104, 65, 246, 15, 21, 84, 157, 14, 118}
```

The next step in our research was examination of the effect of particle population size and the number of iterations of the external cycle on the efficiency of heuristic generation of nonlinear substitutions.

```
Input arguments:
N ← number of blocks in the population
max_itr ← maximum number of iterations
mode ← mode (0 – algorithm from [12], 1 – modified algorithm)
_____________________________________________________
Generation of the initial population of S-boxes:
srand(time(NULL));
int flag = rand()%size;
population ← zeros(2 × N, 256) // 256 for 8 × 8 S-boxes
for i ← 1 to N
     [sbox_i, xr] ← gen_sbox(i+flag)
        Generation occurs using rand () and Fisher-Yates mixing
     population[i] ← sbox_i
end for
population[1] ← aes_sbox
_____________________________________________________
Calculation of nonlinearity for each of the particles:
for i ← 1 to N do:
     NL[i] ← nonlinearity(population[i])
end for
NL_sorted ← sort(NL) // sorting in descending order
gBest ← population[1]; pBest_i ← population[i]; Vel ← zeros(N, 256);
Setting the inertial weight w
_____________________________________________________
The beginning of the improvement phase:
While (max_itr > 0) do:
     Using (3) for w
     c1 ← 2*rand() [0,1]; c2 ← 2*rand() [0,1]; r1 ← rand() [0,1]; r2 ← rand() [0,1];
     NL ← NL_sorted
     for i ← 1 to N do:
             for j = 1 to 256 do:
                      if (mode == 1)
                         Vel[i][j] ← ceil(w*Vel[i][j] + c1*r1*(gBest[i][j] - population[i][j]) +
                      c2*r2*(gBest[j]-population[i][j]))
                      end if
                      if (mode == 0)
                         Vel[i][j] ← ceil(w*Vel[i][j] + c1*r1*(pBest[i][j] - population[i][j]) +
                      c2*r2*(gBest[j]-population[i][j]))
                      end if
                      if (Vel[i][j] < 0)
                              Vel[i][j] ← (Vel[i][j] + 256)mod(256)
                      end if
                      X[i, j] ← int(population[i][j] + Vel[i][j])mod(256)
                      temp_sbox[j] ← X[i, j]
             end for
                Using algorithm from Fig.2 to prevent bijectivity
                if (i == 0)
                     int LAT = LATMax(tempSbox, size, count);
                      int NL = raiseToPower(2, count - 1) - LAT;
                      if (NL > 98)
                              for (int v = 0; v < 15; ++v)
                            srand(tempSbox[v] * (curIter * v) % 256); int coeff = rand() % 50;
                            printf("coeff1 %d", coeff); int coeff2 = rand() % 256;
                            printf("coeff2 %d", coeff2); int temp = tempSbox[coeff];
                            tempSbox[coeff] = tempSbox[coeff2]; tempSbox[coeff2] = temp;
                            end for
                     end if
                end if
             population[N + i] ← temp_sbox
     end for
     for i ← 1 to N do:
             NL_sorted[N + i] ← nonlinearity(population[N + i])
     end for
     NL_sorted ← sort(NL_sorted)
     Discarding of redundant elements
     if (curIter == 0)
            for (int m = 0; m < size; ++m)
                gBest[m] = population[0][m];
            end for
            for (int i = 1; i < N; ++i)
                for (int j = 0; j < size; ++j)
                    pBest[i - 1][j] = population[i][j];
                end for
            end for
      end if
      else
            for (int m = 0; m < size; ++m)
                gBest[m] = population[1][m];
            end for
            for (int i = 2; i < N; ++i)
                for (int j = 0; j < size; ++j)
                    pBest[i - 1][j] = population[i][j];
                end for
            end for
      end else
     Evaluation of all blocks in the population for compliance with the required parameters
     Updating gBest, if find
     max_itr ← max_itr – 1; mode = 0
     ++curIter
end while
Write the final population to file
_____________________________________________________
```

Figure 3: Modified PSO to generate S-Boxes

3. Results of Experimental Research and Discussion

A new modification of PSO was examined in detail during the experiments. The results of test runs to search for blocks with the required parameters are given in Tables 1-13.

Table 1

Experiment 1 (PC 32 cores)

N	MaxIter	M	T, s	Nd	Nd, %
5	10	not found	84.88	–	–
5	50	not found	492.55	–	–
5	100	not found	946.41	–	–
5	150	not found	1353.63	–	–
5	200	not found	1741.28	–	–
10	10	not found	217.65	–	–
10	50	not found	992.38	–	–
10	100	not found	1862.28	–	–
10	150	41	761.95	30	0.8828
10	200	173	2979.45	31	0.8789
20	10	not found	449.93	–	–
20	50	not found	1904.47	–	–
20	100	not found	3584.98	–	–
20	150	not found	5301.96	–	–
20	200	not found	6942.43	–	–
40	10	not found	858.82	–	–
40	50	not found	3682.55	–	–
40	100	not found	7099.77	–	–
40	150	141	9752.98	31	0.8789
40	200	not found	14270.52	–	–

Table 2

Experiment 2 (PC 32 cores)

N	MaxIter	M	T, s	Nd	Nd, %
5	10	not found	104.23	–	–
5	50	not found	538.65	–	–
5	100	not found	1190.65	–	–
5	150	not found	1805.80	–	–
5	200	not found	2314.13	–	–
10	10	not found	238.50	–	–
10	50	not found	1305.35	–	–
10	100	not found	2491.63	–	–
10	150	not found	3573.84	–	–
10	200	not found	4656.12	–	–
20	10	not found	541.57	–	–
20	50	not found	2554.30	–	–
20	100	not found	4785.10	–	–
20	150	128	5721.88	34	0.8671
20	200	180	7902.11	30	0.8828
40	10	not found	1125.40	–	–
40	50	not found	5054.16	–	–
40	100	not found	9507.63	–	–
40	150	not found	13671.93	–	–
40	200	115	9065.45	32	0.8750

Table 3

Experiment 3 (PC 32 cores)

N	MaxIter	M	T, s	Nd	Nd, %
5	10	not found	80.47	–	–
5	50	not found	415.47	–	–
5	100	not found	850.42	–	–
5	150	not found	1265.85	–	–
5	200	not found	1722.77	–	–
10	10	not found	174.28	–	–
10	50	not found	922.19	–	–
10	100	not found	1782.55	–	–
10	150	139	2416.90	30	0.8828
10	200	not found	3433.16	–	–
20	10	not found	370.90	–	–
20	50	not found	1832.72	–	–
20	100	59	1972.26	30	0.8828
20	150	not found	5230.96	–	–
20	200	119	3931.65	33	0.8710
40	10	not found	800.76	–	–
40	50	not found	3621.85	–	–
40	100	not found	7091.61	–	–
40	150	not found	10458.92	–	–
40	200	185	12779.69	31	0.8789

Table 4

Experiment 4 (PC 32 cores)

N	MaxIter	M	T, s	Nd	Nd, %
5	10	not found	77.60	–	–
5	50	not found	390.76	–	–
5	100	not found	808.13	–	–
5	150	not found	1202.86	–	–
5	200	not found	1625.47	–	–
10	10	not found	165.16	–	–
10	50	not found	864.62	–	–
10	100	46	725.23	34	0.8671
10	150	not found	2534.65	–	–
10	200	not found	3364.91	–	–
20	10	not found	359.31	–	–
20	50	not found	1756.20	–	–
20	100	not found	3448.87	–	–
20	150	not found	5132.69	–	–
20	200	not found	6735.40	–	–
40	10	not found	754.74	–	–
40	50	not found	3512.35	–	–
40	100	not found	6878.40	–	–
40	150	not found	10011.85	–	–
40	200	160	10223.11	32	0.8750

Table 5

Experiment 5 (PC 32 cores)

N	MaxIter	M	T, s	Nd	Nd, %
5	10	not found	76.79	–	–
5	50	not found	391.46	–	–
5	100	not found	795.54	–	–
5	150	not found	1213.76	–	–
5	200	not found	1601.32	–	–
10	10	not found	167.58	–	–
10	50	not found	867.30	–	–
10	100	67	1079.23	35	0.8632
10	150	not found	2528.51	–	–
10	200	not found	3370.54	–	–
20	10	not found	356.95	–	–
20	50	not found	1741.53	–	–
20	100	93	3127.57	30	0.8828
20	150	not found	5102.65	–	–
20	200	not found	6763.24	–	–
40	10	not found	753.54	–	–
40	50	not found	3527.99	–	–
40	100	not found	6828.83	–	–
40	150	not found	9957.19	–	–
40	200	128	8043.43	32	0.8750

Table 6

Experiment 6 (PC 32 cores)

N	MaxIter	M	T, s	Nd	Nd, %
5	10	not found	74.34	–	–
5	50	47	354.86	32	0.8750
5	100	60	432.99	31	0.8789
5	150	not found	1193.36	–	–
5	200	25	177.53	33	0.8710
10	10	not found	167.10	–	–
10	50	not found	847.43	–	–
10	100	46	703.24	36	0.8593
10	150	not found	2500.45	–	–
10	200	9	133.74	31	0.8789
20	10	not found	351.87	–	–
20	50	not found	1711.80	–	–
20	100	not found	3380.28	–	–
20	150	87	2739.64	33	0.8710
20	200	not found	6702.15	–	–
40	10	not found	726.33	–	–
40	50	not found	3475.42	–	–
40	100	not found	6769.95	–	–
40	150	not found	10019.90	–	–
40	200	101	6294.79	31	0.8789

Table 7

Experiment 7 (PC 32 cores)

N	MaxIter	M	T, s	Nd	Nd, %
5	10	not found	79.10	–	–
5	50	not found	394.38	–	–
5	100	not found	825.93	–	–
5	150	not found	1236.77	–	–
5	200	not found	1650.94	–	–
10	10	not found	170.88	–	–
10	50	19	293.26	32	0.8750
10	100	not found	1733.82	–	–
10	150	not found	2602.92	–	–
10	200	not found	3467.82	–	–
20	10	not found	373.83	–	–
20	50	not found	1800.63	–	–
20	100	not found	3494.82	–	–
20	150	not found	5237.45	–	–
20	200	not found	6982.87	–	–
40	10	not found	761.22	–	–
40	50	not found	3621.89	–	–
40	100	not found	7057.81	–	–
40	150	not found	10465.29	–	–
40	200	not found	13880.89	–	–

Table 8

Experiment 8 (PC 32 cores)

N	MaxIter	M	T, s	Nd	Nd, %
5	10	7	51.30	32	0.8750
5	50	not found	408.70	–	–
5	100	not found	824.20	–	–
5	150	not found	1258.96	–	–
5	200	not found	1644.69	–	–
10	10	not found	172.86	–	–
10	50	not found	879.15	–	–
10	100	25	392.40	32	0.8750
10	150	not found	2572.43	–	–
10	200	119	1918.16	35	0.8632
20	10	not found	371.46	–	–
20	50	not found	1771.66	–	–
20	100	not found	3531.34	–	–
20	150	59	1909.44	31	0.8789
20	200	59	1899.25	32	0.8750
40	10	not found	755.48	–	–
40	50	not found	3621.89	–	–
40	100	not found	6942.53	–	–
40	150	106	6943.96	32	0.8750
40	200	not found	13843.27	–	–

Table 9

Experiment 9 (PC 32 cores)

N	MaxIter	M	T, s	Nd	Nd, %
5	10	not found	79.37	–	–
5	50	not found	396.52	–	–
5	100	not found	804.42	–	–
5	150	not found	1242.67	–	–
5	200	16	117.86	35	0.8632
10	10	not found	170.65	–	–
10	50	not found	874.37	–	–
10	100	45	723.78	33	0.8710
10	150	not found	2590.71	–	–
10	200	not found	3376.27	–	–
20	10	not found	368.23	–	–
20	50	not found	1812.57	–	–
20	100	not found	3483.90	–	–
20	150	not found	5128.78	–	–
20	200	not found	6737.66	–	–
40	10	not found	760.91	–	–
40	50	not found	3567.87	–	–
40	100	60	3895.49	32	0.8750
40	150	12	784.66	33	0.8710
40	200	122	7741.89	31	0.8789

Table 10

Experiment 10 (PC 32 cores)

N	MaxIter	M	T, s	Nd	Nd, %
5	10	not found	76.87	–	–
5	50	not found	396.38	–	–
5	100	not found	811.63	–	–
5	150	68	503.72	36	0.8593
5	200	not found	1641.91	–	–
10	10	not found	171.87	–	–
10	50	not found	885.23	–	–
10	100	not found	1735.72	–	–
10	150	not found	2597.36	–	–
10	200	not found	3440.48	–	–
20	10	not found	366.27	–	–
20	50	not found	1783.27	–	–
20	100	not found	3530.93	–	–
20	150	not found	5177.53	–	–
20	200	not found	6878.85	–	–
40	10	not found	751.28	–	–
40	50	not found	3600.52	–	–
40	100	not found	7044.17	–	–
40	150	not found	10293.34	–	–
40	200	not found	13629.83	–	–

Table 11

Experiment 11 (PC 32 cores)

N	MaxIter	M	T, s	Nd	Nd, %
5	10	not found	75.93	–	–
5	50	not found	389.89	–	–
5	100	not found	790.66	–	–
5	150	not found	1172.46	–	–
5	200	not found	1609.27	–	–
10	10	not found	165.39	–	–
10	50	not found	856.82	–	–
10	100	81	1326.95	30	0.8828
10	150	not found	2468.59	–	–
10	200	not found	3366.65	–	–
20	10	not found	355.20	–	–
20	50	not found	1744.30	–	–
20	100	not found	3376.48	–	–
20	150	not found	5062.85	–	–
20	200	not found	6741.53	–	–
40	10	not found	740.96	–	–
40	50	5	306.39	32	0.8750
40	100	not found	6799.41	–	–
40	150	not found	10074.13	–	–
40	200	85	5273.26	31	0.8789

Table 12

Experiment 12 (PC 32 cores)

N	MaxIter	M	T, s	Nd	Nd, %
5	10	not found	77.67	–	–
5	50	7	48.54	33	0.8710
5	100	not found	785.60	–	–
5	150	not found	1209.89	–	–
5	200	not found	1615.53	–	–
10	10	not found	165.72	–	–
10	50	not found	868.91	–	–
10	100	73	1192.63	30	0.8828
10	150	not found	2515.39	–	–
10	200	195	3237.48	31	0.8789
20	10	not found	354.79	–	–
20	50	not found	1742.73	–	–
20	100	not found	3437.68	–	–
20	150	not found	5130.86	–	–
20	200	not found	6792.29	–	–
40	10	not found	750.21	–	–
40	50	not found	3545.66	–	–
40	100	not found	6870.33	–	–
40	150	not found	10212.32	–	–
40	200	11	699.60	31	0.8789

Table 13

Experiment 13 (PC 32 cores)

N	MaxIter	M	T, s	Nd	Nd, %
5	10	not found	79.36	–	–
5	50	not found	401.46	–	–
5	100	41	302.12	32	0.8750
5	150	not found	1244.88	–	–
5	200	not found	1615.53	–	–
10	10	not found	169.78	–	–
10	50	not found	886.25	–	–
10	100	not found	1759.42	–	–
10	150	not found	2600.69	–	–
10	200	122	1979.11	32	0.8750
20	10	not found	370.42	–	–
20	50	not found	1833.76	–	–
20	100	not found	3495.82	–	–
20	150	not found	5286.66	–	–
20	200	23	751.50	37	0.8554
40	10	not found	770.72	–	–
40	50	not found	3591.87	–	–
40	100	not found	7075.36	–	–
40	150	not found	10520.47	–	–
40	200	not found	13910.12	–	–

The Tables 1-13 indicate:

- N is number of blocks in the population.
- *MaxIter* is specified number of iterations.
- M is number of iterations to find the required block (or not found).
- T is work time.
- N_d is the number of positions differs from AES.
- N_d, % is coefficient of similarity (in percent) with AES S-Box.

Below is a summary of the information presented in the form of a histogram (see Fig. 4).

In Fig. 4 on the x-axis indicates the number of iterations, on the y-axis is the number of blocks in the population, on z - the number of blocks (with the required parameters) found (as required set: nonlinearity = 104, algebraic immunity = 3, linear redundancy = 0). As you can see, the larger N and *MaxIter* , the more blocks found.

Additionally, the last two columns in Tables 1-13 should be commented on. These columns contain parameters of difference (absolute and relative) from the values of the algebraic S-box of the AES cipher [19,20]. This S-box is used as the initial filling of one of the particles of the swarm (see the line «`population[1] ← aes_sbox`» in Fig. 1, 3), therefore is a certain initial value when generating substitutions. The corresponding differences characterize the "distance" of the found S-box from this initial value.

As you can see, the generated substitution tables are very similar to the AES cipher S-box (almost 90% similarity). S-boxes generated by the algorithm from [21] are also close by this property. Therefore, it is promising to compare these two generation algorithms for the efficiency of computational search.

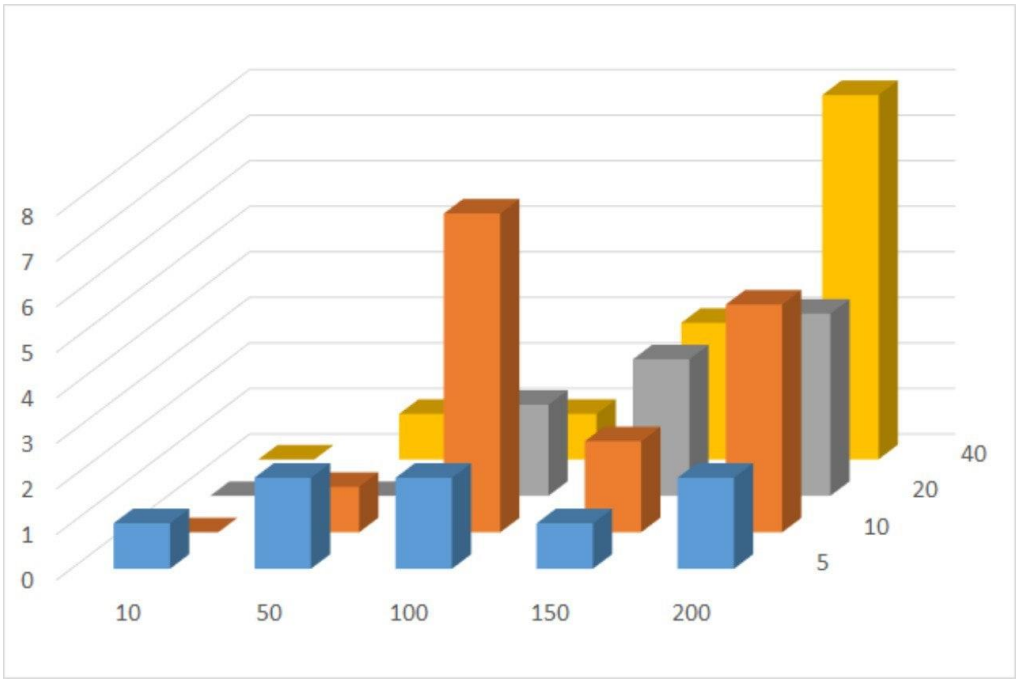

Figure 4: Graph of the dependence of the number of found blocks on N and $MaxIter$

4. Acknowledgements

This work was supported in part by the National Research Foundation of Ukraine under Grant 2020.01/0351.

5. References

[1] B. Schneier, Applied cryptography : protocols, algorithms, and source code in C, New York : Wiley, 1996. http://archive.org/details/appliedcryptogra00schn_328 (accessed July 25, 2020).
[2] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, P.C. van Oorschot, S.A. Vanstone, Handbook of Applied Cryptography, CRC Press, 2018. https://doi.org/10.1201/9780429466335.
[3] S. Rubinstein-Salzedo, Cryptography, Springer International Publishing, Cham, 2018. https://doi.org/10.1007/978-3-319-94818-8.
[4] K. Nyberg, Perfect nonlinear S-boxes, in: D.W. Davies (Ed.), Advances in Cryptology — EUROCRYPT '91, Springer, Berlin, Heidelberg, 1991: pp. 378–386. https://doi.org/10.1007/3-540-46416-6_32.
[5] W. Millan, How to improve the nonlinearity of bijective S-boxes, in: C. Boyd, E. Dawson (Eds.), Information Security and Privacy, Springer, Berlin, Heidelberg, 1998: pp. 181–192. https://doi.org/10.1007/BFb0053732.
[6] J. Álvarez-Cubero, Vector Boolean Functions: applications in symmetric cryptography, 2015. https://doi.org/10.13140/RG.2.2.12540.23685.
[7] D. Souravlias, K.E. Parsopoulos, G.C. Meletiou, Designing bijective S-boxes using Algorithm Portfolios with limited time budgets, Applied Soft Computing. 59 (2017) 475–486. https://doi.org/10.1016/j.asoc.2017.05.052.
[8] J. McLaughlin, Applications of search techniques to cryptanalysis and the construction of cipher components, phd, University of York, 2012. http://etheses.whiterose.ac.uk/3674/ (accessed August 16, 2020).
[9] C. Carlet, Vectorial Boolean functions for cryptography, Boolean Models and Methods in Mathematics, Computer Science, and Engineering. (2006).
[10] A.J. Clark, Optimisation heuristics for cryptology, phd, Queensland University of Technology, 1998. https://eprints.qut.edu.au/15777/ (accessed May 19, 2021).

[11] L.D. Burnett, Heuristic Optimization of Boolean Functions and Substitution Boxes for Cryptography, phd, Queensland University of Technology, 2005. https://eprints.qut.edu.au/16023/ (accessed May 19, 2021).
[12] M. Ahmad, I.A. Khaja, A. Baz, H. Alhakami, W. Alhakami, Particle Swarm Optimization Based Highly Nonlinear Substitution-Boxes Generation for Security Applications, IEEE Access. 8 (2020) 116132–116147. https://doi.org/10.1109/ACCESS.2020.3004449.
[13] A. Kuznetsov, K. Kuznetsova, Comment on "Particle Swarm Optimization Based Highly Nonlinear Substitution-Boxes Generation for Security Applications," in: 2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Cracow, Poland, September 22-25. http://www.idaacs.net (accessed October 15, 2021).
[14] J. Kennedy, R. Eberhart, Particle swarm optimization, in: Proceedings of ICNN'95 - International Conference on Neural Networks, 1995: pp. 1942–1948 vol.4. https://doi.org/10.1109/ICNN.1995.488968.
[15] M.E.H. Pedersen, A.J. Chipperfield, Simplifying Particle Swarm Optimization, Applied Soft Computing. 10 (2010) 618–628. https://doi.org/10.1016/j.asoc.2009.08.029.
[16] Y. Xiaojing, J. Qingju, L. Xinke, Center Particle Swarm Optimization Algorithm, in: 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), 2019: pp. 2084–2087. https://doi.org/10.1109/ITNEC.2019.8729510.
[17] Z. Yimin, S. Guojun, Y. Xiaoguang, Cloud service selection optimization method based on parallel discrete particle swarm optimization, in: 2018 Chinese Control And Decision Conference (CCDC), 2018: pp. 2103–2107. https://doi.org/10.1109/CCDC.2018.8407473.
[18] M.E.H. Pedersen, Tuning & simplifying heuristical optimization, phd, University of Southampton, 2010. https://eprints.soton.ac.uk/342792/ (accessed August 5, 2021).
[19] J. Daemen, V. Rijmen, Rijndael/AES, in: H.C.A. van Tilborg (Ed.), Encyclopedia of Cryptography and Security, Springer US, Boston, MA, 2005: pp. 520–524. https://doi.org/10.1007/0-387-23483-7_358.
[20] J. Daemen, V. Rijmen, Specification of Rijndael, in: J. Daemen, V. Rijmen (Eds.), The Design of Rijndael: The Advanced Encryption Standard (AES), Springer, Berlin, Heidelberg, 2020: pp. 31–51. https://doi.org/10.1007/978-3-662-60769-5_3.
[21] O. Kazymyrov, V. Kazymyrova, R. Oliynykov, A Method For Generation Of High-Nonlinear S-Boxes Based On Gradient Descent, 2013. https://eprint.iacr.org/2013/578 (accessed August 16, 2020).