Finding a Bit-Sliced Representation of 4×4 S-Boxes based on Typical Logic Processor Instructions

Yaroslav Sovyn[1], Ivan Opirskyy[1], and Olha Mykhaylova[1]

[1] *Lviv Polytechnic National University, 12 Stepan Bandera str., Lviv, 79000, Ukraine*

Abstract

The paper is devoted to the development of a method for generating bit-sliced-bioactive descriptions of 4×4 S-Boxes with a reduced number of logic gates. The bit-sliced descriptions generated by the proposed method can improve the performance and security of software implementations of crypto algorithms using 4×4 S-Boxes on various processor architectures (CPU, MCU, GPU). The paper develops a heuristic method for finding a bit-sliced representation that uses typical logical instructions AND, OR, XOR, NOT, AND-NOT, available in most 8/16/32/64-bit processors. Due to the combination of various heuristic techniques in the method (previous calculations, exhaustive search to a certain depth, IDDFS algorithm, refinement search), it was possible to reduce the number of gates in S-Boxes bit-sliced descriptions compared to other known methods. It has been established that the developed method, in 57% of cases, generates a bit-sliced description with fewer gates compared to the best-known methods implemented in the LIGHTER/Peigen utilities if the standard set of processor logical instructions (AND, OR, XOR, NOT) is used. If the processor additionally supports the AND-NOT instruction, then in 54% of cases, it is also possible to generate a bit-sliced description with a smaller number of gates.

Keywords

Bit slicing, 4×4 S-Box, processor instructions, logical minimization.

1. Introduction

The current problem of information protection is to ensure sufficiently high performance of cryptographic algorithms (CA) for a wide class of microprocessor architectures used in various applied tasks [1–3]. In addition, for software implementations of cryptographic algorithms, it is necessary to simultaneously ensure resistance to attacks through side-channel attacks: for low-end CPUs (8/16/32-bit microcontrollers) these are primarily energy consumption analysis attacks, for high-end CPUs (x86, ARM Cortex-A) is a cache attack [4].

To ensure the high performance of crypto algorithms, various approaches to their software implementation are used: the creation of listed tables (Lookup Tables, LUT) for certain operations, integration of hardware crypto accelerators into the processor (for example, AES-NI in x86 processors), application of SIMD technology. 2, AVX-512 in x86-64 CPU), etc. However, these approaches have several disadvantages and limitations and cannot consistently be implemented in a specific processor, especially in low-end processors focused on IoT and embedded systems, which are characterized by limited resources and computing capabilities [5].

Bit slicing [6] is a promising approach that provides a high-performance constant-time implementation of a CA with immunity to time and cache attacks [7], makes maximum use of the capabilities of modern high-end microprocessors to increase performance due to the parallelization of both code execution and data processing and also allows adaptation for low-end CPUs and hardware implementation on FPGAs and ASIC [6]. For many CA, the bit-sliced technology provides the highest speed in software

CQPC-2023: Classic, Quantum, and Post-Quantum Cryptography, August 1, 2023, Kyiv, Ukraine
EMAIL: yaroslav.r.sovyn@lpnu.ua (Y. Sovyn); ivan.r.opirskyi@lpnu.ua (I. Opirskyy); olha.o.mykhailova@lpnu.ua (O. Mykhaylova)
ORCID: 0000-0002-5023-8442 (Y. Sovyn); 0000-0002-8461-8996 (I. Opirskyy); 0000-0002-3086-3160 (O. Mykhaylova)

implementation (if hardware crypto accelerators are not used) for various types of processor architectures [6–15].

The main idea of Bitslicing is to convert CA into a sequence of bit logical operations AND, XOR, OR, NOT, etc. Each such logical operation can be represented in processors by a corresponding instruction, in hardware—by a corresponding gate. The high speed of software Bitslicing is achieved since the CPU processes many cipher elements (bytes, blocks) in parallel, using fast logical instructions and easier execution of some operations (for example, bit permutations, shifts, etc.). The absence of references to precomputed tables in memory and cache and the use of simple logical instructions makes bit-sliced implementations invulnerable to timing and cache attacks and at the same time complicates attacks through third-party channels [16].

To get the maximum speed, you need to minimize the number of logical operations included in the bit-sliced description of the crypto algorithm. Most cryptographic operations produce an unambiguous description when going to a bit-sliced description, or don't give much room for minimization except for non-linear transformations. In CA, nonlinear replacement operations are given in the form $n \times m$ LUT tables, so-called S-Boxes, preferably having size 4×4 ($n = 4$) or 8×8 ($n = 8$) bit. Tables of 4×4 bits are characteristic of both lightweight crypto-algorithms specially designed for efficient implementation on resource-limited processors (e.g. block ciphers PRINCE, LED, Piccolo, hash functions PHOTON, Spongent) and general-purpose crypto algorithms (e.g. block symmetric ciphers Serpent, Twofish, hash functions BLAKE, Whirlpool) [17].

The main problem with the bit-sliced implementation of the CA is to represent the S-Box with the minimum possible number of logic gates/instructions. This problem is NP-complete and admits an exact solution only for very simple cases ($n \leq 3$ and some $n = 4$). Therefore, most modern methods and utilities for generating bit-sliced descriptions of S-Boxes use heuristic approaches. Given the number of gates, this does not guarantee that the resulting solution is optimal. However, they provide a much better result compared to the universal methods for minimizing logical functions (for example, the Karnaugh map method or the Quine-McCluskey method of simple implicants). Therefore, the problem of finding the optimal bit-sliced representation even for small S-Boxes (4×4) is far from being solved, which requires the search for new heuristic approaches, one of which is presented in our work.

2. Bit-Sliced Implementation

The most difficult stage in the bit-sliced implementation, which largely determines the speed in general, is the logical representation of tables of non-linear substitution of S-Boxes. In the case of hardware implementation, logic gates (Gate Equivalent, GE) {AND, OR, XOR, NOT} act as the logical basis, in software bit-sliced implementation, the gates are replaced by corresponding instructions that are present in most processor architectures. Therefore, in the future, we will use the concepts of valve and instruction as synonyms. It should be noted that some processors do not have the NOT instruction, which is emulated by the XOR instruction. Since the logic instructions of the processor mainly process two operands, the logic elements must also be two-input (Fig. 1) so that one can unambiguously pass from the logical representation to the software one.

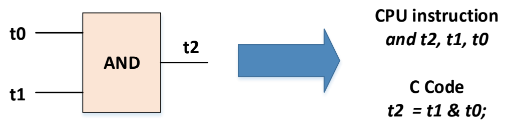

Figure 1: The transition from logical to programmatic bit-sliced representation

The bit-sliced approach to cryptographic representation was first proposed by E. Biham in [6] to speed up the software implementation of the DES cipher. In the same paper, the algorithm of bit-sliced representation of DES S-Boxes (6×4) with logic gates XOR, AND, OR, NOT is described, for which, on average, one DES S-Box requires 100 gates.

In [16], M. Kwan proposed a much more efficient approach to finding a bit-sliced representation using DES S-Boxes as an example. It treats each S-Box output bit as a function of the six input bits, represented by a Karnaugh map, and placed in a 64-bit variable. All input and intermediate variables can also be considered as 6-bit Karnaugh maps described by 64-bit numbers. Then the task is formulated as follows: it is necessary to combine the existing input and intermediate maps in such a way as to obtain the desired output variable. One input variable acts as a selector combining the functions of five variables. To find the representation of functions of five variables with the minimum number of gates, brute force is used, and the gates are found in the previous steps. Depending on the order in which the search will be performed, there are 6! available options for input variables and 4! options for output variables. This gives a total of 17280 search options, among which the option with the minimum number of gates is selected. As a result, the average number of gates for a bit-sliced description of one DES S-Box has decreased from 100 to 56.

M. Kwan's algorithm with some improvements is implemented in the form of the sboxgates utility, which generates a bit-sliced description for arbitrary S-Boxes up to 8×8 inclusive [17]. This utility allows you to specify an arbitrary set of two-input gates, use LUT-like ternary logic instructions that have become available in GPUs and x86-CPUs with AVX-512 support, specify the number of iterations of the search algorithm, parallelize the search between processor cores, etc.

SAT-Solvers programs can be used to minimize S-Boxes. These programs are designed to effectively solve the feasibility problem of Boolean formulas (SATisfiability problem, SAT). The object of the SAT problem is a Boolean formula consisting only of constants (0/1), variables, AND, OR, and NOT operations. The problem is as follows: can all variables be assigned the values False and True so that the formula becomes True? Specialized SAT-Solvers programs, built on efficient solution algorithms, accept a set of equations as input and output the result in the form of SAT if a solution is found and UNSAT if no solution is found. To find a logic circuit with a given number of gates, you can form an equation where the variables specify all possible connections between gates and operations and try to solve them with the help of SAT-Solvers. The advantage of this approach is that if a solution with n gates (SAT) is found and UNSAT is obtained for $n - 1$ gates, then we are guaranteed to have found the minimum possible bit-sliced description.

In [18], SAT-Solvers were used to find the bit-sliced representation of 4-bit S-Boxes, and some of the results are presented in Table 1 [14], where the Bitslice Gate Complexity (BGC) criterion denotes the optimal solution with the minimum number of gates/operations.

Table 1
SAT minimization of S-Boxes by the BGC criterion

S-Box	Size $n\times m$	Bit Slice Gate Complexity
Prost	4×4	8 (4 AND, 4 XOR)
Piccolo/ Piccolo^{-1}	4×4	10 (1 AND, 3 OR, 4 XOR, 2 NOT)
Lac	4×4	11 (2 AND, 2 OR, 6 XOR, 1 NOT)
Rectangle	4×4	$\in \{10,11,12\}$ (4 OR, 7 XOR, 1 NOT)
Rectangle^{-1}	4×4	$\in \{11,12\}$ (1 AND 3 OR, 7 XOR, 1 NOT)
Minalpher	4×4	$\geq$ 11

Data in Table 1 should be interpreted as follows. For example, for the S-Box of the Piccolo cipher, it was possible to find a bit-sliced representation of ten gates with the help of SAT-Solvers and to prove that the representation from BGC = 9 does not exist (UNSAT) and, therefore, BGC(Piccolo) = 10. For the S-Box of the Rectangle cipher, it was not possible to represent from BGC = 12, which means that there is no solution with BGC = 10 or 11. For the Minalpher cipher, it was not possible to find bit-sliced descriptions at all, but it was only possible to prove that solutions from BGC = 10 do not exist.

So, the problem with SAT-Solvers is that they don't always find solutions for "heavy" S-Boxes, such as Minalpher, which may require more than 12–13 gates. For relatively simple S-Boxes with 11–13 gates, SAT-Solvers cannot always prove that the found representation is minimal, as can be seen in the Rectangle example. In addition, the disadvantage of this method is poor scalability: the SAT approach only works for small S-Boxes, up to 5×5 in size, however, for 8×8 S-Boxes, this approach cannot be implemented in terms of computational complexity.

In [19], using SAT-Solvers to minimize S-Boxes is also proposed. The difference in the approach is that initially with the help of SAT-Solvers, they find the logical representation of the S-Box according to the criterion of Multiplicative Complexity (MC) that is, a representation containing the minimum possible number of non-linear gates (AND). Thus, the logical representation of the S-Box is divided into two parts: non-linear (AND gates) and linear (XOR, NOT gates), after which the linear part is minimized separately, also using SAT-Solvers.

This approach is characterized by all the shortcomings of the previously considered approach: finding the Multiplicative Complexity is also an NP-complete problem that can be solved with the help of SAT-Solvers for relatively simple S-Boxes, poor scalability. In addition, although the solution at each of the two steps is optimal, this does not guarantee that the overall solution is also optimal.

[20] describes the open-source utility LIGHTER, which is currently the most effective utility for finding the bit-sliced description of 4×4-bit S-Boxes. LIGHTER can flexibly specify a set of two- and three-inlet valves and their weighting factors, which are taken into account during minimization. This allows more realistic optimization in the case of hardware implementation, when different logic gates differ in crystal area, power consumption, delay, etc., due to the consideration of these parameters in the weighting factors. When logical instructions are equivalent for software implementation, setting the same weighting coefficients for all gates is enough. The LIGHTER search algorithm itself combines two approaches: search using the breath-first-search algorithm and the meet-in-the-middle strategy. Two graphs are built: one starts from the base vectors and searches forward, and the other starts from the desired vectors and searches back. Both graphs move towards each other using the given logical operations until they meet. Next, a path is selected that combines these two graphs with the minimum cost, taking into account the weighting factors for each gate. The utility demonstrates high time efficiency compared to SAT methods, and its results, which, although cannot be considered optimal, are quite close to the results obtained by SAT utilities and are much better than the results of the sboxgates utility.

The paper [20] describes the Peigen open-source utility (Platform for Evaluation, Implementation, and Generation of S-boxes), which allows you to find bit-sliced descriptions of S-Boxes in various logical bases, applying specified minimization criteria for hardware and software implementations. The Peigen utility can evaluate the cryptographic properties of S-Boxes, generate S-Boxes according to specified criteria, and search for an optimized representation of S-Boxes according to certain criteria, in particular, according to BGC, MC criteria, etc. The search algorithms for the bit-sliced description of the Peigen utility are based on the algorithms from the LIGHTER utility [21], but their temporal efficiency has been improved, in particular, enumerations and several additional techniques have been used. However, even with the improvements made, the utility only works effectively with 4-bit S-Boxes.

Generating an optimized bit-sliced CA implementation requires a significant amount of time to write and debug code and requires a good knowledge of processor architecture, low-level tools, and optimization techniques at the hardware and software levels. Therefore, in [21], a high-level Usuba language is presented, which allows for describing a symmetric cryptographic primitive, and the Usuba compiler itself will generate a highly optimized, parallelized, and vectorized bit-sliced code. However, to generate a bit-sliced S-Box description, either a simple minimization algorithm is used, which gives a far from the optimal result, or a ready-made optimized description is taken from the database included in Usuba if the S-Box is present in it. Thus, description generation for the S-Box is a weak point of the bit-sliced compiler Usuba.

2.1. Research Objective

The purpose of this paper is to present a method for generating a bit-sliced description of 4×4 S-Boxes, which provides better results compared to existing ones, which will increase the speed and security of hardware and software implementations of a wide range of cryptographic algorithms using S-Boxes of a given type.

Features of S-Boxes representation for bit-sliced implementation

In the CA specifications, S-Boxes are preferably defined as LUT. For example, the 4×4 S-Box of the PRESENT cipher has the form shown in Table 2. In Bit-sliced, LUT-Tables are considered logical functions given by truth tables. For example, the S-Box PRESENT cipher will look shown in Table 3.

Table 2
S-Box LUT table of the PRESENT cipher

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S(x)	12	5	6	11	9	0	10	13	3	14	15	8	4	7	1	2

Table 3

Bit-sliced-oriented S-Box representation of the PRESENT cipher

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Hex
x_0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	*0xff00*
x_1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	*0xf0f0*
x_2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	*0xcccc*
x_3	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	*0xaaaa*
$S(x)$	12	5	6	11	9	0	10	13	3	14	15	8	4	7	1	2	—
y_0	1	0	0	1	1	0	1	1	0	1	1	1	0	0	0	0	*0x0ed9*
y_1	1	1	1	0	0	0	0	1	0	1	1	0	1	1	0	0	*0x3687*
y_2	0	0	1	1	0	0	1	0	1	1	1	0	0	1	0	1	*0xa74c*
y_3	0	1	0	1	1	0	0	1	1	0	1	0	0	1	1	0	*0x659a*

So, a compact representation of the S-Box in the form of a truth table will look like: $S(x) = y$, where $x = \{x_0, x_1, x_2, x_3\} = \{0xff00, 0xf0f0, 0xcccc, 0xaaaa\}$ is input bit-sliced variables, $y = \{y_0, y_1, y_2, y_3\} = \{0x0ed9, 0x3687, 0xa74c, 0x659a\}$ – the output bit-sliced variables that define a specific substitution table, and the 16-bit numbers that specify x and y will be called vectors.

We are looking at the bit-sliced representation of the S-Box for two typical sets of logical instructions, which are most commonly used in processor command systems:

● Standard set (STD), consisting of instructions NOT, AND, OR, XOR. This instruction set is supported by almost any 8/16/32/64-bit processor and is universal.

● The extended set (EXT), in addition to the instructions of the standard set (NOT, AND, OR, XOR), additionally contains the AND-NOT ($c = \overline{a} \& b$) instruction, which is present in some processors, for example, with the x86-64 or ARM architecture.

The task of searching for a bit-sliced S-Box representation by the BGC criterion can be formulated as follows: given four base vectors base = $\{x_0, x_1, x_2, x_3\}$, y previous calculations ou need to find the vectors y = $\{y_0, y_1, y_2, y_3\}$ using the minimum number of logical instructions from the given set of STD or EXT.

2.2 Previous Calculations

At the precalculation stage, certain data is found and stored once, which is then repeatedly used in our bit-sliced description search algorithm. This data is of two types:

1. For each 16-bit vector V, BGC (V) is a minimal number of GE valves required to represent it, the so-called "complexity" of the vector.

Since vectors are represented by 16-bit numbers, there are 65536 vectors in total, four of them are base vectors base = $\{x_0–x_3\}$ and two are logical constants const = {*0x0000, 0xffff*} for which BGC is 0, so there are 65530 vectors whose complexity needs to be estimated. In Table 4 shows the found distribution of vectors by their BGC value for the STD and EXT instruction sets.

Table 4

Distribution of 16-bit vectors over BGC

BGC	0	1	2	3	4	5	6	7	8
Number of vectors (STD)	6	22	126	691	3181	12639	27165	19670	2036
Number of vectors (EXT)	6	34	258	1465	6549	17404	24596	13864	1360

As seen in Table 4, the maximum complexity is 8, which means that any 16-bit vector can be represented with at most 8 gates. This gives an upper bound for the bit-sliced complexity of an arbitrary S-Box described by four vectors $y_0–y_3$, equal to 32 gates.

2. Building LUT tables to represent all graphs at a given depth.

Furthermore, LUT tables were built containing all possible combinations that can be formed using a given number of *ge* operations from the STD/EXT instruction set. These sequences of vectors we shall call called graphs. Tables are formed by sequentially using the GEN_TABLE function, which takes an $n×m$ table as input and returns $n_{\text{new}}×(m + 1)$ table containing all possible combinations formed by a given set of instructions from input table rows.

LUT tables is used in the search algorithm to speed up the selection of candidate graphs in the first step. Thus, for the set of instructions STD, a table q_5 was built containing all possible 118491958 graphs to a depth of 6 instructions (ge = 6), and for the set EXT, a table $q4$ was built containing 21832210 graphs to a depth of 5 instructions (ge = 5). Further construction of the listed tables is impractical since it will require too much memory.

The obtained step-by-step results are presented in the Table 5.

Table 5
Properties of LUT tables q_0–q_5

Table	q_0 (ge = 1)	q_1 (ge = 2)	q_2 (ge = 3)	q_3 (ge = 4)	q_4 (ge = 5)	q_5 (ge =6)
Dimension STD	22×1	429×2	8593×3	186434×4	4462108×5	118491958×6
Dimension EXT	34×1	927×2	24899×3	706608×4	21832210×5	—

3. Bit-Sliced Implementation

3.1 Bit-Sliced Representation Search Algorithm

At the top level of the search algorithm, iterates over all values y_0-y_3, generates each of them from the listed LUT table of the matrix of candidate graphs gr_i = $FIRST\_STEP(y_i)$, and passes them to the depth-first search algorithm $FIND\_BS(gr_i)$. The FIND_BS depth-first search algorithm finds the remaining values in an attempt to use a minimum of gates and returns the constructed augmented graph matrices gr_0–gr_3. From the results obtained, graphs with the minimum BGC value are selected (Fig. 2).

Thus, the search algorithm performs four iterations, starting from different values y. Note this initial value for y_{start}. At the stage gr_i = $FIRST\_STEP(y_{\text{start}})$, using the LUT-table q, a matrix of graphs gr_i, is generated, containing all possible graphs with vector y_{star} at a certain gate depth d_{start}. Depending on which BGC group the u-start vector belongs to, heuristically selected d-start values are presented in Table 6 in order to ensure acceptable calculation time and amount of required memory.

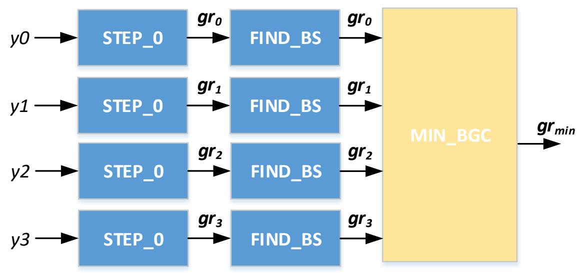

Figure 2: Generalized structure of the S-Box bit-sliced description search algorithm

Depending on which BGC group the u-start vector belongs to, heuristically selected d-start values are presented in the Table 6 in order to ensure acceptable calculation time and amount of required memory. If, for example, $bgc(y_0)$ = 1, then the graph matrix gr_0 after FIRST_STEP will contain all graphs with a length of 6 gates (d_{start} = 6) in which the vector y_0 occurs.

Table 6
Depth of generating graphs containing y_{start} in the FIRST_STEP

BGC-group y_{start}	1	2	3	4	5	6	7	8
d_{start}	6	6	7	8	8	8	8	9

Further, all graphs of the candidate in gr_i are sorted into three groups: gr_1y, gr_2y, gr_3y with the same number of vectors in each group graph are 1, 2, and 3, respectively. Note this number y_find. Further, the search is carried out for each non-empty group separately in accordance with Fig. 3.

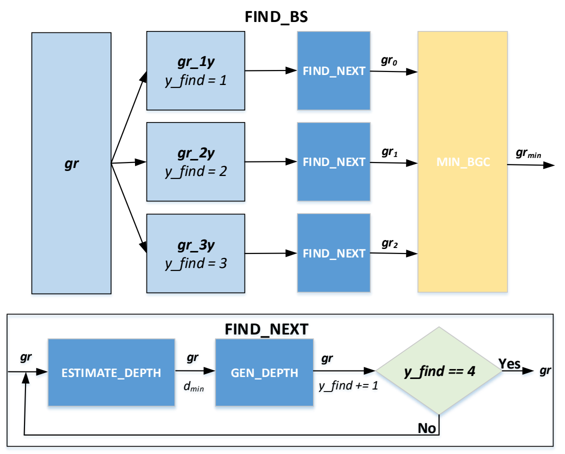

Figure 3: Generalized search scheme for bit-sliced representations by the FIND_BS algorithm

The FIND_NEXT algorithm searches y_i, ui one by one until all four values y_0-y_3 are found. The graph matrix *gr* is given as an input in the form of an $n \times m$ table, each row of which contains *y_find* values from the set $\{y_0–y_3\}$. Each row of the table stores *m* vectors explicitly and vectors $x_0–x_3$ implicitly.

First, the minimum distance $d_{\min}$ is estimated for group gr, at which the nearest unfound value y_x is located among all graphs ESTIMATE_DEPTH. For this, the fast FAST_FIND function of comprehensive forward search to a given depth of 1/2/3/4 steps has been developed. The search and selection of options are carried out using the algorithm of depth-first search with iterative deepening—Iterative Deepening Depth-First Search (IDDFS).

If in the set *gr* at all search depths (1/2/3/4) not a single value y_x is found ($d_{\min} \geq 5$), then a step forward is made and a new table of size $n_{\text{new}} \times (m + 1)$, is generated from the table *gr* using the GEN_TABLE function, after which the search is repeated, etc. (Fig. 4). After the estimate $d_{\min} \leq 4$ is found, using the GEN_DEPTH algorithm, the transition is made from the set of graphs from *y_find* = *n_y* to the set of graphs from *y_find* = *n_y + 1*.

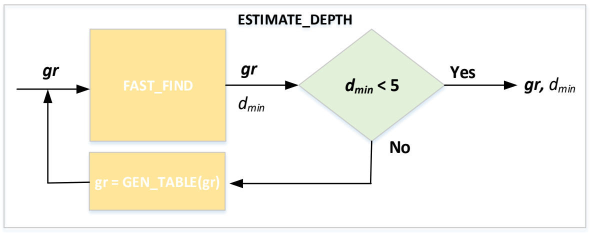

Figure 4: Estimation of the search depth to find the next vector y_x

For each of the *gr* groups, the graphs are selected to find the values of $d_{\min}$ and for the $gr_{\min}$ group, run ahead *gr* = *GEN_TABLE*($gr_{\min}$). For the generated set *gr*, the graphs with the known values $d = d_{\min} - 1$ are selected again, for them to fight ahead and so far, until d becomes equal to 0. After that, only those graphs are selected into the group, to avenge n_y + 1 value y. These steps are repeated until all the *y* values are found.

The FIND_BS algorithm at each step estimates the minimum distance $d_{\min}$, at which the nearest value of y_x is located, and generates the corresponding graphs. As shown in Fig. 5 this route starts with graphs containing y_a, generated using FIRST_STEP, from which the nearest value y_b is located at a distance of d_{ab} gates then we go to y_c located at the minimum distance d_{bc} from y_b and at the distance d_{cd} we find the last vector y_d.

However, the movement with minimal steps along the trajectory from the vector y_a to y_d does not always give the optimal result in general (although this is the case in most cases). There may be a situation where the choice of the minimum value of *d* in the first steps leads to large values of *d* in the following steps and, as a result, to a non-optimal logical representation. For example, let's assume that in the first step we got $d_{ab} = 1$, in the second $d_{bc} = 4$, and in the third $d_{cd} = 3$, that is, the route will be a total of 8 gates (Fig. 5), but it is possible that if in the first step, we followed a different route and graphs with $d_{ab} = 2$ were selected, then in the second step we could find the value of y_c with $d_{bc} = 3$ and in the third y_d with $d_{cd} = 2$, and we would get a shorter total route with 7 valves. Consequently, the second route resulted in a bit-sliced representation with a lower BGC value.

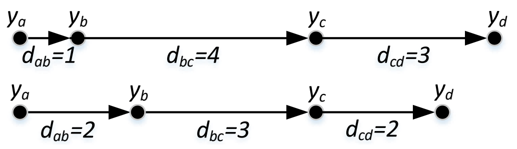

Figure 5: Finding the bit-sliced description for different routes

In order to take into account different possible routes in the search algorithm, refining searches are carried out according to the scheme presented in Fig. 6. If we have a set of graphs containing 3 out of 4 possible values of y, then the search for the fourth value is always carried out at the minimum possible depth $d_{\min}$ (*SEARCH_3Y*). For graphs with two values in *y* (*y_find* = 2), the third value is searched for by two routes: $d_{\min}$ and $d_{\min} + 1$, after which the SEARCH_3Y search is applied to the found graphs with *y_find* = 3. For graphs with one value in *y* (*y_find* = 1), the search for the second value takes place along three routes: $d_{\min}$, $d_{\min} + 1$ and $d_{\min} + 2$, after which the SEARCH_2Y search is applied to the found graphs with *y_find* = 2.

3.2 Results and Discussion

The method proposed in the work was implemented in the Python language, and to ensure speed, the main data processing functions are implemented based on the numpy and pyopencl libraries.

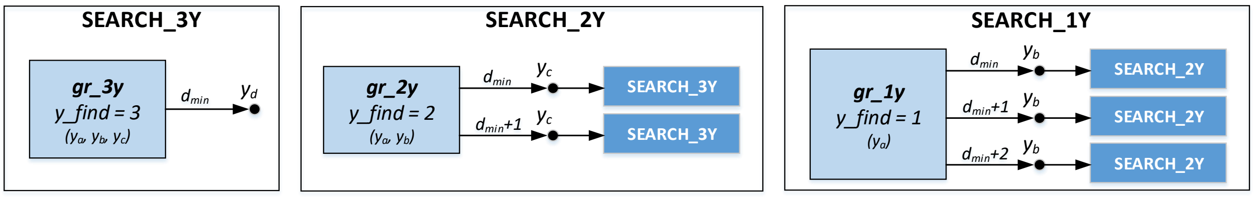

Figure 6: Refinement search scheme in the FIND_BS algorithm

To evaluate our algorithm, 225 4×4 S-Boxes of various cryptographic algorithms were taken. We used the open-source projects LIGHTER and PEIGEN to obtain a BGC score for selected S-Boxes and compare it with our results. Bit-sliced descriptions of S-Boxes obtained by our method are available at the link [22].

The results are presented in Table 7. Column data in the table should be interpreted as follows:

LUT is a tabular representation of the S-Box, where the line '0123456789abcdef' should be understood as $S(x)$ = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15.

BSL representation of S-Box in bit-sliced format. The line '0ed9_3687_a74c_659a' should be understood as follows: y_0 = 0x0ed9, y_1 = 0x3687, y_2 = 0xa74c, y_3 = 0x659a.

CY is BGC of vectors y_0–y_3. The line '6285' should be interpreted as: BGC(y_0) = 6, BGC(y_1) = 2, BGC(y_2) = 8, BGC(y_3) = 5.

R is the results, contain the BGC value obtained using the method described in the article.

L/P contains the BGC value obtained using the LIGHTER/PEIGEN utilities [15, 16]. These utilities use the same search algorithm, but due to optimizations, they can sometimes give different results for the same S-Box, in these cases, the minimum value was chosen.

S-Boxes that have a higher BGC value compared to the one obtained by our method are marked in red, and those that have the same BGC value as our results are marked in yellow.

Table 7
Comparison of BGC for different S-Boxes

S-Box	LUT	BSL	STD			EXT		
			CY	R	L/P	CY	R	L/P
Piccolo	e4b238091a7f6c5d	cd94_1e1d_fc03_aaa5	6533	10	11	6533	10	10
Piccolo^{-1}	68341eca5792df0b	b714_aaa5_3369_b4e2	7346	10	11	6345	10	10
Lac	e9f0d4ab128376c5	9996_3ac5_f035_44d7	3566	11	11	3566	11	11
Prost	048f15e927acbd63	b2b8_d748_6a6a_3ccc	5622	8	8	5622	8	8
Rectangle	65ca1e79b03d8f42	2dd2_a569_6867_39ac	4466	12	12	3466	12	12
Rectangle^{-1}	94fae106c7382b5d	e625_369c_c396_a91d	7437	12	12	7437	12	12
Minalpher	b34128cf5de069a7	a38b_d493_97c4_66e1	7876	15	16	7866	15	16
SKINNY	c6901a2b385d4e7f	cd94_e1e2_fc03_aaa5	6533	11	11	6433	10	10
TWINE	c0fa2b9583d71e64	1ee4_6a3c_ec85_256d	6577	15	15	5577	15	15
PRINCE	bf32ac916780e5d4	62c7_131f_f322_5473	7557	16	18	7546	16	17
Lucifer_S0	cf7aedb026319458	5c66_075e_6237_907b	5657	15	17	5557	14	15
Lucifer_S1	72e93b04cd1a6f85	a639_3837_b385_6b2c	7577	16	18	7577	16	16
PRESENT	c56b90ad3ef84712	659a_a74c_3687_0ed9	4777	14	14	3677	14	14
PRESENT^{-1}	5ef8c12db463079a	69a5_ad46_2697_c19e	4786	14	14	4786	14	14
JH_S0	904bdc3f1a26758e	31d9_9ec8_b8b4_c2b9	7658	16	16	7648	15	15
JH_S1	3c6d5719f204bae8	11f9_7325_493e_f18a	7766	16	18	6766	16	16
Iceberg_S0	d7329ac1f45e60b8	4597_592e_1f43_c971	7678	15	16	7578	15	15
Iceberg_S1	4afc0d9be6173582	3ce4_9b86_2b2d_41ee	6764	15	15	5764	14	15
Luffa	de015a76b39cf824	1759_53e2_98d3_3d23	7786	13	14	7686	12	13
Noekeon	7a2c48f0591e3db6	7741_d847_a959_6a6a	6852	12	12	6852	12	12
Hb1_S0	865f1ca9eb2470d3	d29c_974a_592e_43e9	6666	15	16	6656	14	15
Hb1_S1	07e15b823ad6fc49	953a_1ba6_7c16_b664	6766	14	15	5766	14	15
Hb1_S2	2ef5c19ab468073d	e16c_6587_a61e_89d6	6666	16	16	6666	15	15
Hb1_S3	0734c1afde6b2895	c9a6_1ec6_879a_6bd0	6666	14	16	6665	14	15
Hb1_S0^{-1}	d4afb21c07695e83	9a59_a63c_368b_689d	6587	15	16	6587	14	15
Hb1_S1^{-1}	0378e4b16f95da2c	1ec6_6356_9b34_b658	6666	15	15	6566	14	15
Hb1_S2^{-1}	c50e93adb6784f12	65b2_a768_368b_29d9	6686	16	16	6686	15	15
Hb1_S3^{-1}	05c23fa1de6b4897	c9b2_8e78_9726_6b64	6676	14	16	6675	14	15
Hb2_S0	7ce9215fb6d048a3	85e9_c395_16c7_658e	7686	15	16	7686	15	16

Hb2_S1	4a168f7c30ed59b2	7964_c56a_1ce9_6cb2	7576	15	16	7576	15	15
Hb2_S2	2fc156ade8340b97	e49a_a563_89b6_63c6	7665	15	15	7665	14	15
Hb2_S3	f4589721a30e6cdb	c2b5_9b61_7827_e919	7766	15	16	7766	14	15
Hb2_S0^{-1}	b54fc690d3e81a27	934b_e629_853e_2d59	7777	16	16	6767	15	16
Hb2_S1^{-1}	92f80c364d1e7ba5	b645_78c6_9ba4_6a2d	7667	16	17	6667	15	15
Hb2_S2^{-1}	c30ab45f9e6d2781	a9d2_369a_2ee1_4b99	6666	15	15	6666	14	15
Hb2_S3^{-1}	a76912c5348fdeb0	599a_6927_3ac6_7c49	5768	15	16	5758	14	15
DES_S0_0	e4d12fb83a6c5907	b16c_8771_9c27_2ae5	5766	15	16	5766	14	15
DES_S0_1	0f74e2d1a6cb9538	78c6_4b36_265e_9d52	6676	13	14	6676	13	13
DES_S0_2	41e8d62bfc973a50	5d92_39e4_4b35_279c	6586	14	15	5585	14	15
DES_S0_3	fc8249175b3ea06d	87e1_5e89_c993_9a27	6868	15	15	6868	15	15
DES_S1_0	f18e6b34972dc05a	4b63_8679_5a99_992d	6656	14	14	6656	13	14
DES_S1_1	3d47f28ec01a69b5	e41b_58b9_919e_69d2	5855	15	15	5855	14	14
DES_S1_2	0e7ba4d158c6932f	b1cc_e81e_8d66_965a	5653	12	12	4653	12	12
DES_S1_3	d8a13f42b67c05e9	a539_47b4_6e61_c927	6666	15	17	6566	15	16
DES_S2_0	a09e63f51dc7b428	1be4_5879_2ed8_964d	4777	15	16	4777	15	16
DES_S2_1	d709346a285ecbf1	e41b_69d2_5c63_7a89	5567	14	15	5567	14	15
DES_S2_2	d6498f30b12c5ae7	9369_e562_d827_6939	6755	13	15	6755	13	14
DES_S2_3	1ad069874fe3b52c	3aa5_5e92_a794_9666	6553	13	14	6553	13	13
DES_S3_0	7de3069a1285bc4f	994b_92ad_e827_b4c6	6776	15	16	6776	14	16
DES_S3_1	d8b56f03472c1ae9	92ad_66b4_4b39_e827	7677	14	17	7577	14	16
DES_S3_2	a690cb7df13e5284	17e4_2d63_99d2_49b5	6757	15	16	6757	14	16
DES_S3_3	3f06a1d8945bc72e	2d63_e81b_b64a_99d2	7765	15	16	7765	14	16
DES_S4_0	2c417ab6853fd0e9	9e58_4cf1_5a96_d962	6745	15	16	6735	14	15
DES_S4_1	eb2c47d150fa3986	35e2_9c27_8579_6c4b	7677	15	16	7677	15	15
DES_S4_2	421bad78f9c5630e	2b6c_b15a_9d61_87b8	7575	16	17	7575	16	16
DES_S4_3	b8c71e2d6f09a453	ca99_9369_63ac_1aa7	6657	15	16	6657	15	15
DES_S5_0	c1af92680d34e75b	e61a_b46c_7a49_929d	6666	15	16	6566	15	15
DES_S5_1	af427c9561de0b38	66d2_691b_0db6_ac63	6766	15	17	5756	15	16
DES_S5_2	9ef528c3704a1db6	718d_c996_a54e_6867	7566	16	16	7556	15	16
DES_S5_3	432c95fabe17608d	8d72_1bc6_9a69_c3d8	4665	14	14	4565	13	13
DES_S6_0	4b2ef08d3c975a61	9d92_691e_5a99_26da	5556	14	15	5556	14	15
DES_S6_1	d0b7491ae35c2f86	266d_b38c_ad19_69a5	7584	14	14	7484	14	14
DES_S6_2	14bdc37eaf680592	626d_87e4_26da_4b9c	6666	15	16	6666	14	16
DES_S6_3	6bd814a7950fe23c	4b96_78c3_9aa5_994e	5555	14	14	5555	13	14
DES_S7_0	d2846fb1a93e50c7	96e1_8d72_d839_4b65	6477	14	15	6477	14	14
DES_S7_1	1fd8a374c56b0e92	4a67_ac72_27c6_691e	8765	15	16	8765	15	15
DES_S7_2	7b419ce206adf358	781b_36c3_5a65_9c72	7555	14	15	7555	13	13
DES_S7_3	21e74a8dfc90356b	b58a_d12d_639c_87e4	5646	14	15	5636	14	14
Serpent_S0	38f1a65bed42709c	52cd_19b5_9764_c396	8863	14	14	7763	14	14
Serpent_S1	fc27905a1be86d34	6359_568d_b44b_2e93	7848	14	14	7848	14	14
Serpent_S2	86793cafd1e40b52	639c_a4d6_4da6_25e9	4766	13	13	3766	13	13
Serpent_S3	0fb8c963d124a75e	63a6_b4c6_e952_913e	6667	15	15	5666	14	15
Serpent_S4	1f83c0b6254a9e7d	d24b_69ca_e692_b856	6675	15	15	6675	14	15
Serpent_S5	f52b4a9c03e8d671	d24b_662d_7493_1ce9	6687	15	15	6687	14	15
Serpent_S6	72c5846be91fd3a0	3e89_69c3_196d_5b94	8486	14	14	8486	14	14
Serpent_S7	1df0e82b74ca9356	7187_a9d4_c716_1cb6	7676	16	16	7675	15	16
Serpent_S0^{-1}	d3b0a65c1e47f982	3947_9a36_1ee1_7295	8648	14	14	8648	14	14
Serpent_S1^{-1}	582ef6c3b4791da0	3d91_45bc_2679_695a	7774	14	14	7673	14	14
Serpent_S2^{-1}	c9f4be12036d58a7	9a56_c6b4_9c2d_6837	4676	13	13	4666	13	13
Serpent_S3^{-1}	09a7be6d35c248f1	c39a_497c_56e8_64b6	5766	15	15	5766	14	15
Serpent_S4^{-1}	5083a97e2cb64fd1	e469_2dd8_7ac1_66b4	7576	15	15	7575	14	15
Serpent_S5^{-1}	8f2941deb6537ca0	1d6a_5b86_36d2_61cb	5666	15	15	5666	14	15
Serpent_S6^{-1}	fa1d536049e72c8b	8a3d_9c63_2d59_e60b	7478	14	14	7477	14	14
Serpent_S7^{-1}	306d9ef85cb7a142	2d59_9c65_4b6c_16f8	7765	16	16	7765	15	16
GOST_1	4a92d80e6b1c7f53	f614_b38a_7991_2ab6	6686	15	16	6686	15	15
GOST_2	eb4c6dfa23810759	ea62_23d3_607d_84eb	5577	16	16	5577	15	16
GOST_3	581da342efc7609b	ca2d_9bb0_1f49_c71a	8776	16	18	8676	16	16
GOST_4	7da1089fe46cb253	d0cb_b585_4f83_19e6	7554	15	16	7554	14	14
GOST_5	6c715fd84a9e03b2	647c_ea25_0977_4ee2	7677	17	19	6676	16	17
GOST_6	4ba0721d36859cfe	59d2_c336_ea91_f486	6577	15	17	5476	15	16
GOST_7	db413f590ae7682c	08fb_5e32_9c65_a6a3	5775	14	16	5675	14	15
GOST_8	1fd057a4923e6b8c	2537_3e62_98b6_e946	6676	16	16	6676	15	16

LBlock_0	e9f0d4ab128376c5	9996_3ac5_f035_44d7	3566	11	11	3566	11	11
LBlock_1	4be9fd0a7c562813	c53a_9996_0f35_22be	4366	11	11	4365	11	11
LBlock_2	1e7cfd06b593248a	0f35_9996_22be_c53a	6364	11	11	6354	11	11
LBlock_3	768b0f3e9acd5241	9969_22eb_5ca3_0fac	4655	11	11	4655	11	11
LBlock_4	e5f072cd1849ba63	9996_f035_44d7_3ac5	3665	11	11	3665	11	11
LBlock_5	2dbcfe097a631845	9996_0f35_c53a_22be	3646	11	11	3645	11	11
LBlock_6	b94e0fad6c573812	5ca3_9969_0fac_22eb	5456	11	11	5456	11	11
LBlock_7	daf0e49b218375c6	3ac5_9996_f035_44d7	5366	11	11	5366	11	11
LBlock_8	87e5fd06bc9a2413	c53a_9996_22be_0f35	4366	11	11	4356	11	11
LBlock_9	b5f0729d481cea36	44d7_f035_9996_3ac5	6635	11	11	6635	11	11
SC2000_4	25ac7f1bd609483e	49f2_c2b5_933a_a9ac	6755	15	16	5754	15	16
MIBS	4f38dac0b57e2619	c716_3d26_2e53_897a	7786	17	17	7686	16	17
KLEIN	74a91fb0c3268ed5	c279_2e65_e923_716c	7777	17	17	7777	16	17
Panda	0132fc9ba6875ed4	58d6_2b9c_fa30_65f0	7655	14	15	7644	13	13
MANTIS	cad3ebf789150246	0eec_a0fa_c8d5_0377	6465	13	14	5465	13	14
GIFT	1a4c6f392db7508e	1ee1_8d72_9a3c_c6aa	4454	11	11	4454	11	11
UDCIKMP11	086d5f7c4e2391ba	7878_ce64_03fc_d2aa	2424	8	8	2424	8	8
Luffa_v1	7dbac4835f60912e	3387_c68d_8733_925e	5666	13	13	5655	12	12
Enocoro_S4	139a5e72d0cf486b	8957_c8ea_5d70_ad2c	7567	16	17	7466	15	16
Qarma_sigma0	0e2a9f8b6437dc15	dcb0_0dae_bb22_30fa	6646	14	14	6544	13	14
Qarma_sigma1	ade6f735980cb124	31f2_507d_88be_1b17	6656	15	15	5656	14	15
Qarma_sigma2	b68fc09e3745d21a	5b49_a38b_1e9a_90dd	7757	16	17	7756	16	16
Midori_Sb0	cad3ebf789150246	0eec_a0fa_c8d5_0377	6465	13	14	5465	13	14
Midori_Sb1	1053e2f7da9bc846	0dcd_8af8_d1d4_3f50	6556	15	17	5555	15	17
Anubis_S0	d7329ac1f45e60b8	4597_592e_1f43_c971	7678	15	16	7578	15	15
Anubis_S1	4afc0d9be6173582	3ce4_9b86_2b2d_41ee	6764	15	15	5764	14	15
Khazad_P	3fe054bcda967821	9553_5a47_19b6_27c6	6666	15	17	6666	15	15
Khazad_Q	9e56a23cf04d7b18	7945_317a_1d8e_a993	6766	16	17	6666	16	16
Fox_S1	2519eac8647fdb03	bc0e_ad31_1f52_38f8	7875	16	17	6765	15	16
Fox_S2	b41f03eda875c296	4cad_a569_9cca_53c9	7467	13	13	7457	13	13
Fox_S3	dab14389572cf06e	13ad_d626_db11_98c7	7577	16	18	7577	16	17
Whirlpool_E	1b9cd6f3e874a250	44d7_35e2_4d78_135e	6777	16	18	6776	16	16
Whirlpool_R	7cbde49f638a2510	62cd_1b95_21bb_0cde	6866	15	16	6765	15	16
SMASH_256_S1	6dc7f13a8b5024e9	867a_52d9_641f_c396	6883	14	14	6773	14	14
SMASH_256_S2	1b60ed5ac29738f4	5c63_5a96_c974_65b2	6476	13	13	6376	13	13
SMASH_256_S3	429c81e7f50b6a3d	cba4_79c2_93c9_a95c	7665	15	15	7665	14	15
CS_cipher_G	a602be18d453fc79	dd50_583b_7722_b1b1	5834	13	14	4734	13	13
GOST2_1	6af43850de712bc9	ad54_3617_474d_e326	6667	16	17	6666	16	17
GOST2_2	e0817a56d2493fcb	b958_b2b1_65d1_e925	7666	16	17	7666	15	16
Magma_1	c462a5b9e8d703f1	ece0_695c_4d27_47d1	4676	16	17	4676	15	16
Magma_2	68239a5c1e47bd0f	b958_9a2d_aec1_b2b2	7774	16	17	7774	16	16
Magma_3	b3582fade174c960	26a7_4573_5da4_31e9	7767	16	16	7767	16	16
Magma_4	c821d4f670a53e9b	d958_b5c4_29f1_e453	7788	15	15	6678	15	15
Magma_5	7f5a816d093eb42c	16a7_5c4b_a8c7_9a9a	8773	15	16	8772	15	15
Magma_6	5df692cab78143e0	2b17_63ac_524f_45d6	6576	15	17	6576	15	16
Magma_7	8e25691cf4b0da37	d568_e516_939a_35a3	7656	16	17	7656	15	16
Magma_8	17ed05834fa69cb2	52ab_ce86_2b2e_764c	7666	16	17	7656	16	17
CLEFIA_SS0	e6ca872fb14059d3	f3a0_81eb_54a7_619d	5767	15	17	4767	15	15
CLEFIA_SS1	640d2ba39cef8751	e9a8_2cf1_6e0b_1f68	6675	15	16	6675	14	15
CLEFIA_SS2	b85ea64cf72310d9	db05_0f39_43ec_c19b	6557	16	16	6557	15	16
CLEFIA_SS3	a26d345e0789bfc1	ba58_3297_62ec_7c89	6667	15	16	6667	15	16
Golden_S0	035869c7dae41fb2	71a6_e692_2dd4_6768	6766	15	15	6765	14	15
Golden_S1	03586cb79eadf214	59c6_36d2_9ab4_1f68	6665	15	15	6665	14	15
Golden_S2	03586af4ed9217cb	b646_a972_63d4_c768	5766	15	16	5766	14	14
Golden_S3	03586cb7a49ef12d	b4c6_59d2_9ab4_9d68	6666	14	16	6566	14	14
Twofish_Q0_T0	817d6f320b59eca4	0e6e_52f4_b43c_7a29	6648	15	16	5538	15	16
Twofish_Q0_T1	ecb81235f4a6709d	d1d4_1d65_9b83_c50f	5676	16	17	5676	15	17
Twofish_Q0_T2	ba5e6d90c8f32471	cc65_5c1b_653c_076b	5767	15	16	5757	15	15
Twofish_Q0_T3	d7f4126e9b3085ca	2717_86e6_60cf_d385	6658	14	15	6658	14	15
Twofish_Q1_T0	28bdf76e31940ac5	873c_21f5_c8f8_649e	5656	16	17	4646	15	16
Twofish_Q1_T1	1e2b4c376da5f908	3ac9_15ce_1bb2_b62a	6766	15	16	6666	14	15
Twofish_Q1_T2	4c75169a0ed82b3f	e45c_f2a4_862f_aec2	7576	15	17	6576	15	16
Twofish_Q1_T3	b951c3de647f208a	0c6f_9da1_0fd4_c8d3	6757	16	17	6757	15	16

Serpent_type_S0	03567abcd4e9812f	a956_c47a_879c_9de0	3767	13	13	3666	13	13
Serpent_type_S1	035869a7bce21fd4	71a6_2dd2_e694_6768	6476	12	12	6375	12	12
Serpent_type_S2	035869b2d4e1af7c	6966_74d2_e714_b568	4666	12	13	3556	11	12
Serpent_type_S3	03586af4ed9217cb	b646_a972_63d4_c768	5766	15	16	5766	14	14
Serpent_type_S4	03586cb79eadf214	59c6_36d2_9ab4_1f68	6665	15	15	6665	14	15
Serpent_type_S5	03586cb7a49ef12d	b4c6_59d2_9ab4_9d68	6666	14	16	6566	14	14
Serpent_type_S6	03586cb7ad9ef124	36c6_59d2_9ab4_1f68	4665	12	12	4565	12	12
Serpent_type_S7	03586cb7dae41f29	b1c6_66d2_2db4_a768	6656	13	13	5556	11	13
Serpent_type_S8	03586cf1a49edb27	b4c6_e952_9a74_3d68	6656	14	15	6655	13	14
Serpent_type_S9	03586cf2e9b7da41	9e46_2dd2_5974_3768	7455	13	13	7355	12	12
Serpent_type_S10	03586df19c2ba74e	29e6_bc52_e274_9b68	6656	13	14	6656	12	12
Serpent_type_S11	03586df274eba19c	6966_1dd2_8774_dc68	4666	12	13	3556	12	12
Serpent_type_S12	03586df2c9a4be17	d266_b4d2_a974_3768	5455	12	13	5455	11	12
Serpent_type_S13	03586fa179e4bcd2	53a6_9572_6d34_7668	6766	14	15	5766	14	15
Serpent_type_S14	0358749ef62badc1	a956_1f92_63b4_79c8	3767	13	13	3666	13	13
Serpent_type_S15	035879beadf4c261	8676_65d2_5e94_17e8	6675	14	14	5675	13	14
Serpent_type_S16	03589ce7adf46b12	6696_b5c2_1ee4_2778	4665	12	14	3655	12	13
Serpent_type_S17	0358ad94f621cb7e	6966_e712_d3a4_b178	4665	12	13	3565	11	12
Serpent_type_S18	0358bc6fe9274ad1	ca96_2dd2_59e4_63b8	6477	13	13	6377	13	13
Serpent_type_S19	035a7cb6d429e18f	a956_94da_93b4_d968	3767	13	13	3767	13	13
BLAKE_1	ea489fd61c02b753	f170_b8a3_62e5_127b	6786	16	17	6786	15	16
BLAKE_2	b8c052fdae367194	74d1_1f61_9ad4_43c7	5876	15	16	5876	15	15
BLAKE_3	7931dcbe265a40f8	445f_4bc5_56b1_c8f2	6886	16	17	5785	15	17
BLAKE_4	905724afe1bc683d	c68d_55d8_99ac_adc1	6558	15	16	6558	15	15
BLAKE_5	2c6a0b834d75fe19	dea0_34ad_3f06_b26a	6766	16	16	6756	15	16
BLAKE_6	c51fed4a0763928b	9a2e_ae98_067b_d0b9	5577	15	16	5577	15	15
BLAKE_7	db7ec13950f4862a	05e7_e44e_2d1d_949b	7456	15	15	7456	14	15
BLAKE_8	6fe9b308c2d714a5	9c3a_4a37_ad07_459e	6857	14	16	6757	14	15
BLAKE_9	a2847615fb9e3cd0	57d0_1b33_69b8_6f05	6566	15	17	6566	15	16
GOST_IETF_1	96328b17a4efc0d5	c8e5_0dae_de82_5d31	7667	16	17	7567	16	17
GOST_IETF_2	37e98af0526cb4d1	d14b_1667_6d46_587c	7877	15	15	7876	15	15
GOST_IETF_3	e462b3d8cf5a0719	e670_2a3d_2747_8bd1	6776	16	17	6776	15	17
GOST_IETF_4	e7acd13902b4f856	54f2_9647_d81b_349d	6768	15	16	6768	15	16
GOST_IETF_5	b5198df0e423c7a6	286f_ed41_b362_5179	7767	16	16	7767	15	16
GOST_IETF_6	3adc120b75948fe6	2795_e1a3_eb0c_748e	8766	13	14	7766	13	14
GOST_IETF_7	1d297a608c45f3be	781b_f074_9e52_d32a	7557	15	17	7456	15	16
GOST_IETF_8	baf50ce8623917d4	7c0d_2747_e16c_48e7	7766	15	16	7766	16	16
Kuznyechik_nu0	253b69ea04f18dc7	ac2e_84dd_e652_74e8	6676	17	18	6676	16	17
Kuznyechik_nu1	76c90f8145bed23a	56a9_ec23_1b27_9c6c	4553	11	12	4553	10	11
Kuznyechik_sigma	cd048bae3952f167	b722_d9e0_d48b_12f3	5676	15	16	5676	15	16
Optimal_S0	012d47f68bc93ea5	9a6a_72e4_a4f8_6f48	3567	13	14	3566	13	14
Optimal_S1	012d47f68be359ac	3a6a_4ee4_94f8_e748	4465	12	13	4465	11	12
Optimal_S2	012d47f68be3ac59	ca6a_1ee4_64f8_b748	4673	12	13	4563	12	13
Optimal_S3	012d47f68c53aeb9	cc6a_78e4_26f8_f348	5664	14	15	4564	14	14
Optimal_S4	012d47f68c9bae53	cc6a_b8e4_62f8_3f48	5674	14	15	4664	14	15
Optimal_S5	012d47f68cb9ae35	cc6a_74e4_a2f8_3f48	5654	15	16	4654	14	15
Optimal_S6	012d47f68cb9ae53	cc6a_b4e4_62f8_3f48	5574	14	15	4464	14	14
Optimal_S7	012d47f68ceba935	e86a_5ce4_86f8_3f48	5774	14	15	5674	14	14
Optimal_S8	012d47f68e95ab3c	6c6a_72e4_8af8_b748	6553	12	12	5553	12	12
Optimal_S9	012d47f68eb359ac	3c6a_4ee4_92f8_e748	6465	14	15	5465	13	14
Optimal_S10	012d47f68eb5a93c	6c6a_56e4_8af8_b748	6653	15	15	5653	14	15
Optimal_S11	012d47f68eba59c3	b46a_8ee4_52f8_6f48	6647	15	16	6646	14	15
Optimal_S12	012d47f68eba93c5	b46a_2ee4_c2f8_5f48	6766	15	16	6665	14	15
Optimal_S13	012d47f68ec95ba3	b86a_e2e4_16f8_6f48	5657	15	15	5656	14	15
Optimal_S14	012d47f68ecb395a	786a_9ae4_46f8_af48	6766	15	15	5766	15	15
Optimal_S15	012d47f68ecb93a5	b86a_6ae4_86f8_5f48	5776	15	16	5675	15	15
Num1_DL_04_0	0bc5619a3ef8d427	956a_c792_b61c_1ec6	3556	12	14	3556	12	13
Num1_DL_04_1	0cda5be7f6213894	59b4_17e8_83d6_616e	6556	12	13	6555	11	13
Num1_DL_13_0	0c9761f23b4ed8a5	936c_4bd8_9c5a_7a46	3656	12	12	3656	11	12
Num1_DL_13_1	0c97f2613b4ea5d8	639c_1b78_6c5a_da16	4656	12	12	3655	11	12
Num1_DL_13_2	0b85fc36e47921da	6c5a_95d2_47b8_c936	5643	12	12	5643	11	12
Num1_DL_13_3	0d4b7e926a3581fc	6c5a_47b8_c936_d26a	5436	11	12	5435	11	12
Num1_DL_22_0	0d82eb75f63c419a	65e2_8778_1bd2_c936	6363	11	11	6363	11	11

Num1_DL_22_1	0be1a7d46c9f5832	5c6a_c936_1be4_2e56	6346	11	11	6346	11	11
Num1_DL_22_2	0b69c53ed7842af1	c36a_72c6_4bb4_659a	5644	11	11	5633	11	11
Num1_DL_22_3	0e95f8a73b6c41d2	639c_87d2_5c9a_4a76	4466	11	11	3466	11	11
mCrypton_S0	4f38dac0b57e2619	c716_3d26_2e53_897a	7786	17	17	7686	16	17
mCrypton_S1	1c7a6d53fb20849e	43e5_879c_a176_d32a	8677	16	18	8676	16	17
mCrypton_S2	7ec209da3f5864b1	c761_538b_3647_4ae6	7877	16	17	7776	16	17
mCrypton_S3	b0a7d642ce3915f8	7c19_46ad_6378_cb15	8867	17	18	8867	16	17
Σ				3190	3349		3097	3231

In general, as the results in Table 7, our method showed better results than the LIGHTER/PEIGEN utilities for both the STD and EXT instruction sets. It provides a bit-sliced description with fewer gates for 129 S-Boxes out of 225 (57.3%) using the standard processor logic instruction set and for 123 S-Boxes out of 225 (54.7%) using the extended instruction set. The total number of gates for the bit-sliced description of all 225 S-Boxes in our method is less by 5.0% and 4.3% for the STD and EXT instruction sets, respectively.

The LIGHTER/PEIGEN utilities did not generate a bit-sliced description with fewer instructions for any S-Box than obtained by our method, and the maximum difference in the number of instructions for an S-Box description is 3 and 2 for the STD and EXT sets, respectively.

It should also be noted that the developed method for 'simple' S-Boxes (BGC $\leq$ 12) generates the smallest possible description indicating the same results as those obtained using SAT-Solvers.

4. Conclusion

The paper presents a method for generating a bit-sliced description of arbitrary 4×4 bijective S-Boxes, focused on software implementations on any 8/16/32/64-bit processors that support AND, OR, XOR, NOT, AND-NOT instructions. To date, the method proposed in the paper is the most effective method known to us according to the BGC criterion, which is confirmed by the research results presented in the work. The method combines heuristic techniques at various stages of searching a bit-sliced representation, in particular: recalculation, exhaustive search to a depth of up to four gates, IDDFS algorithm for searching and cutting options, and refining search. If necessary, the developed approach can be adapted to support additional logical instructions.

5. References

[1] I. Opirskyy, Y. Sovyn, O. Mykhailova, Heuristic Method of Finding Bitsliced-Description of Derivative Cryptographic S-Box, in IEEE 16th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (2022) 104–109. doi: 10.1109/TCSET55632.2022.9766883.

[2] Y. Sovyn, et al., Minimization of Bitsliced Representation of 4×4 S-Boxes based on Ternary Logic Instruction, in Cybersecurity Providing in Information and Telecommunication Systems, vol. 3421 (2023) 12–24.

[3] S. Yevseiev, et al., Development of Niederreiter Hybrid Crypto-Code Structure on Flawed Codes, Eastern-European Journal Of Enterprise Technologies, 1(9) (2019) 27–38. doi: 10.15587/1729-4061.2019.156620

[4] V. Buriachok, V. Sokolov, P. Skladannyi, Security Rating Metrics for Distributed Wireless Systems, in: Workshop of the 8th International Conference on "Mathematics. Information Technologies. Education:" Modern Machine Learning Technologies and Data Science, vol. 2386 (2019) 222–233.

[5] I. Kuzminykh, et al., Investigation of the IoT device lifetime with secure data transmission, Internet of Things, Smart Spaces, and Next Generation Networks and Systems, vol. 11660 (2019) 16–27. doi: 10.1007/978-3-030-30859-9_2

[6] E. Biham, A Fast New DES Implementation in Software, in International Workshop on Fast Software Encryption (1997) 260–272.

[7] E. Kasper, P. Schwabe, Faster and Timing-Attack Resistant AES-GCM, in 11th International Workshop Cryptographic Hardware and Embedded Systems (2009) 1–17.

[8] A. Adomnicai, T. Peyrin, Fixslicing AES-Like Ciphers: New Bitsliced AES Speed Records on ARM-Cortex M and RISC-V, IACR Transactions on Cryptographic

Hardware and Embedded Systems, 1 (2021) 402–425.

[9] P. Schwabe, K. Stoffelen, All the AES You Need on Cortex-M3 and M4, in International Conference on Selected Areas in Cryptography (2016) 180–194.

[10] J. Zhang, M. Ma, P. Wang, Fast Implementation for SM4 Cipher Algorithm based on Bit-Slice Technology, in International Conference on Smart Computing and Communication (2018) 104–113.

[11] N. Nishikawa, H. Amano, K. Iwai, Implementation of Bitsliced AES Encryption on CUDA-enabled GPU, in International Conference on Network and System Security (2017) 273–287.

[12] S. Matsuda, S. Moriai, Lightweight Cryptography for the Cloud: Exploit the Power of Bitslice Implementation, in International Workshop on Cryptographic Hardware and Embedded Systems (2012) 408–425.

[13] M. Kwan, Reducing the Gate Count of Bitslice DES, IACR Cryptology ePrint Archive, 51 (2000).

[14] M. Dansarie, Sboxgates: A Program for Finding Low Gate Count Implementations of S-Boxes, Journal of Open Source Software, 6(62) (2021) 1–3.

[15] K. Stoffelen, Optimizing S-Box Implementations for Several Criteria Using SAT Solvers, in 23rd International Conference on Fast Software Encryption (2016) 140–160.

[16] F. Kipchuk, et al., Assessing Approaches of IT Infrastructure Audit, in: IEEE 8th International Conference on Problems of Infocommunications, Science and Technology (2021). doi: 10.1109/picst54195.2021.9772181

[17] V. Sokolov, P. Skladannyi, H. Hulak, Stability Verification of Self-Organized Wireless Networks with Block Encryption, in: 5th International Workshop on Computer Modeling and Intelligent Systems, vol. 3137 (2022) 227–237.

[18] N. Courtois, T. Mourouzis, D. Hulme, Exact Logic Minimization and Multiplicative Complexity of Concrete Algebraic and Cryptographic Circuits, International Journal On Advances in Intelligent Systems, 6(3,4) (2013) 165–176.

[19] J. Jean, et al., Optimizing Implementations of Lightweight Building Blocks, IACR Transactions on Symmetric Cryptology, 4, (2017) 130–168.

[20] Z. Bao, et al., Peigen—A Platform for Evaluation, Implementation, and Generation of S-boxes, IACR Transactions on Symmetric Cryptology (2019) 330–394.

[21] D. Mercadier, Usuba, Optimizing Bitslicing Compiler, PhD Thesis, Sorbonne University, France (2020).

[22] Y. Sovyn, Bitsliced 4x4 S-Boxes 2023 (2023). https://drive.google.com/drive/|folders/1vK2ng__UiVmk-cQAUzDOxS-X1x3DZp1T?usp=drive_link