Minimization of Bitsliced Representation of 4×4 S-Boxes based on Ternary Logic Instruction

Yaroslav Sovyn[1], Volodymyr Khoma[1], Ivan Opirskyy[1], and Valerii Kozachok[2]

[1]*Lviv Polytechnic National University, 12 Stepan Bandera str., Lviv, 79000, Ukraine*
[2]*Borys Grinchenko Kyiv University, 18/2, Bulvarno-Kudryavska str., Kyiv, 04053, Ukraine*

Abstract

The article is devoted to methods and tools for generating software-oriented bit-sliced descriptions of bijective 4×4 S-Boxes with a reduced number of instructions based on a ternary logical instruction. Bitsliced descriptions generated by the proposed method make it possible to improve the performance and security of software implementations of crypto-algorithms using 4×4 S-Boxes on various processor architectures. The paper develops a heuristic minimization method that uses a ternary logical instruction, which is available in ×86–64 processors with AVX-512 support and some GPU processors. Thanks to the combination of various heuristic techniques (preliminary calculations, exhaustive search to a certain depth, refinement search) in the method, it was possible to reduce the number of gates in bit-sliced descriptions of S-Boxes compared to other known methods. The corresponding software in the form of a utility in the Python language was developed and its operation was tested on 225 S-Boxes of various crypto-algorithms. It was established that the developed method generates a bit-sliced description with a smaller number of ternary instructions in 90.2% of cases, compared to the best-known method implemented in the sboxgates utility.

Keywords

Bit-slicing, ternary logic instruction, 4×4 S-Box, CPU, logic minimization, software implementation, sboxgates, speed.

1. Introduction

Given the ever-increasing volumes and speeds of data processing, a very important requirement for Cryptographic Algorithms (CA) is to provide sufficiently high performance for a wide class of microprocessor architectures [1, 2]. The no less important aspect for software implementation of cryptographic algorithms is the increased resistance to side-channel attacks: for low-end CPUs (8/16/32-bit microcontrollers) these are primarily energy consumption analysis attacks, and for high-end CPUs (×86, ARM Cortex-A) are primarily time and cache attacks. To ensure the high performance of crypto-algorithms, various approaches to their software implementation are used. This includes the creation of precomputed tables (Lookup Tables, LUT) for certain operations, the integration of hardware crypto-accelerators into the processor (e.g. AES-NI in ×86 processors), the use of SIMD technology for parallelization of the encryption process (e.g. SSE/AVX2/AVX-512 vector instructions in ×86–64 CPU), the use of GPU computing power, etc. However, all these approaches have several limitations and cannot always be implemented in a specific processor.

Bit-slicing [3] is one of the promising approaches that provide a high-performance constant-time implementation of a CA with immunity to time and cache attacks [4]. It makes the most 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 CPU and hardware implementation on FPGA and ASIC. For many CAs, it is the bit-sliced approach

CPITS 2023: Workshop on Cybersecurity Providing in Information and Telecommunication Systems, February 28, 2023, Kyiv, Ukraine
EMAIL: yaroslav.r.sovyn@lpnu.ua (Y. Sovyn); volodymyr.v.khoma@lpnu.ua (V. Khoma); ivan.r.opirskyi@lpnu.ua (I. Opirskyy); v.kozachok@kubg.edu.ua (V. Kozachok)
ORCID: 0000-0002-5023-8442 (Y. Sovyn); 0000-0001-9391-6525 (V. Khoma); 0000-0002-8461-8996 (I. Opirskyy); 0000-0003-0072-2567 (V. Kozachok)

that provides the highest speed in software implementation (if hardware crypto accelerators are not used) for various types of processor architectures [3–9]. The absence of references to precomputed tables in memory and cache, as well as data-dependent conditional transitions, makes bit-sliced implementations invulnerable to time and cache attacks, at the same time complicating attacks through third-party channels.

The basic idea of bit-slicing is to convert a cryptographic algorithm into a sequence of bit logical operations of the type AND, XOR, OR, NOT. In processors, each such logical operation can be represented by a corresponding instruction, and in hardware—by a corresponding gate (we will use the concepts of gate and instruction as synonyms in this work). The high speed of software bit-slicing is achieved because 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.). For software-oriented bit-sliced implementations, in addition to classic ones, it is possible to use more complex logical instructions supported by a certain processor, and thus reduce their total number. So, for example, many processors support the AND-NOT instruction (×86–64, ARM), some NOR and NAND (ARM), etc. [10].

In order 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 generate a single-valued description when going to a bit-sliced description or do not give much room for minimization except for nonlinear transformations. In CA, nonlinear replacement operations are specified in the form of $n \times m$ LUT-tables, so-called S-Boxes, which are mostly 4×4 ($n = 4$) or 8×8 ($n = 8$) bits in size. Tables of 4×4 bits are typical for both lightweight crypto algorithms specially designed for efficient implementation on resource-constrained processors (e.g. block ciphers PRINCE, LED, Piccolo, hash functions PHOTON, Spongent) and general purpose crypto algorithms (e.g. block symmetric ciphers Serpent, Twofish, Magma, hash functions BLAKE, Whirlpool).

Thus, the main resource for increasing performance in the bit-sliced implementation of the CA is the representation of S-Boxes with the minimum possible number of logic gates/instructions. This problem is NP-complete and allows an exact solution only for very simple cases ($n \leq 3$ and some $n = 4$), so most modern methods and utilities for generating bit-sliced S-Boxes use heuristic approaches that do not guarantee that the obtained solution is optimal but provide a much better result compared to universal methods of minimizing logical functions (for example, the method of Carnot maps as well as the method of simple Quine-McCluskey implicants).

In addition to traditional dual-operand logic instructions, some processors support the ternary logic instruction *ternary logic (a, b, c, imm8*), which allows calculating an arbitrary Boolean function from the three operands *a*, *b*, *c* specified by the truth table in the 8-bit variable *imm8* (Fig. 1).

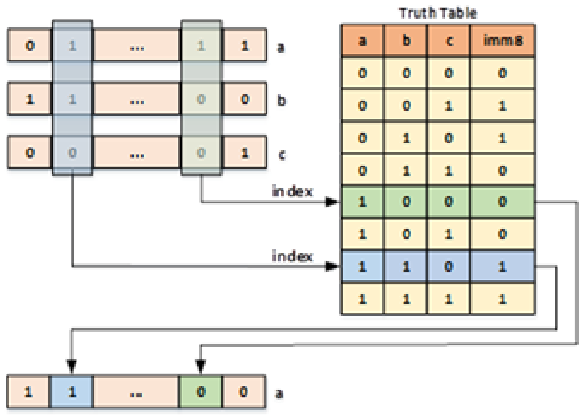

Figure 1: The principle of operation of the ternary instruction *ternary logic (a, b, c, imm8)*

The Ternary Instruction (TI) is present in the following processors:

- ×86–64 with support for 512-bit SIMD instructions from the AVX-512F extension. Since the *vpternlogq zmm_a, zmm_b, zmm_c, imm8* instruction is included in the AVX-512F (Foundation) basic extension, it is supported by all processors with AVX-512 technology.
- some GPU processors. For example, on an Nvidia GPU, this instruction has the form *lop3.b32 d, a, b, c immLut*, which calculates over the 32-bit operands *lop3.b32 d, a, b, c immLut*, the logical function given by the truth table in *immLut* and stores the result in *d*.

One ternary instruction can replace several dual-operand logical instructions, so its use allows generating a bit-sliced description with a significantly smaller number of operations, and therefore increasing the speed of program implementation. However, in this case, even for 4-bit S-Boxes, finding a guaranteed optimal representation is mostly impossible and it is necessary to use heuristic minimization methods. However, the existing heuristic minimization methods are mostly focused on the use of dual-operand instructions, so they cannot be directly used to generate a bit-sliced description based on TI.

Therefore, the problem of finding the optimal bit-sliced representation based on the ternary instruction even for 4×4 S-Boxes is far from being solved, which requires the search for new heuristic approaches, one of which is presented in our work.

2. Review of Literary Sources

Let us analyze the methods and means for finding the bit-sliced description of 4×4 S-Boxes according to the Bitslice Gate Complexity (BGC) criterion, which denotes the optimal solution with the minimum number of operations.

The bit-sliced approach to cryptographic representation was first proposed by E. Biham in [1] to speed up the software implementation of the DES cipher. In the same work, E. Biham described the algorithm of bit-sliced representation of DES S-Boxes (6×4) by logic gates XOR, AND, OR, NOT. In the algorithm, from six input variables, two input variables have been selected that form all possible combinations with the help of given logic gates, from which four output variables are then constructed. On average, with a bit-sliced description using this method, one DES S-Box requires 100 gates.

In [8], 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 Carnot map, and placed in a 64-bit variable. All input and intermediate variables can also be considered as 6-bit Carnot 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, an exhaustive search (brute force) is used, and the gates are found in the previous steps. Depending on the order in which the search is carried out, 6! options are available for input variables, and 4! Options—for output variables. This gives a total of 17,280 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 decreased from 100 to 56.

To minimize S-Boxes, the SAT-Solvers programs can be used, designed to effectively solve the feasibility problem of Boolean formulas (SATfeasibility problem, SAT). The object of the SAT problem is a Boolean formula consisting only of constants (0/1), variables, and 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 produce 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 would specify all possible connections between gates and the operation 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.

SAT-Solvers were used in the works [9, 10] to find the bit-sliced representation of some 4-bit S-Boxes. In general, the problem with SAT-Solvers is that they do not always find solutions for "heavy" S-Boxes that 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.

The work [11] 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 gates 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. For a software implementation, when logical instructions are equivalent, it is enough to set the same weighting coefficients for all gates.

The LIGHTER search algorithm itself combines two approaches: searching using the Breath-First-Search (BFS) algorithm and the Meet-In-The-Middle (MITM) strategy. That is, two graphs are built: one starts from the base vectors and performs a forward search, and the other starts from the searched vectors and performs a backward search. Both graphs move toward each other using the given logical operations until they meet. Next, a path is selected that connects these two graphs with the minimum weight that takes into account the weighting factors for each gate. The utility demonstrates high time efficiency compared to SAT methods, and its results, although they cannot be considered optimal, are quite close to the results obtained by SAT utilities.

In [12], the open-source utility Peigen (Platform for Evaluation, Implementation, and Generation of S-boxes) is described, which makes it possible to find a bit-sliced description of S-Boxes in various logical bases, applying the specified minimization criteria for hardware and software implementations. Peigen's bit-sliced description search algorithms are based on algorithms from the LIGHTER utility, but their time efficiency has been improved, in particular, recalculation 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 implementation of the CA requires considerable time spent on writing and debugging the code and requires a good knowledge of processor architecture, low-level tools, and optimization techniques at the hardware and software levels. Therefore, in [13], the high-level Usuba language is presented, which makes it possible to describe a symmetric cryptographic primitive, and the Usuba compilator itself will generate a highly optimized, parallelized, and vectorized bit-sliced code. However, to generate the bit-sliced description of the S-Box, 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 S-Box is a weak point of the Usuba bit-sliced compiler.

The considered methods and utilities form a bit-sliced description using mainly two-input logic elements and do not support a ternary logic instruction. The only utility known to us today for generating bit-sliced descriptions of S-Boxes based on the ternary instruction is sboxgates [14]. This open-source utility implements the M. Kwan algorithm with some improvements and is able to generate a bit-sliced description for arbitrary S-Boxes up to and including 8×8. Many optimizations of M. Kwan's algorithm in *sboxgates* are borrowed from the SBOXDiscovery project, which was intended exclusively for generating bit-sliced descriptions of DES S-Boxes. The utility allows you to specify an arbitrary set of two-input gates, use a ternary logic instruction, specify the number of iterations of the search algorithm, parallelize the search between processor cores, etc. [15,16]. In the case of 4×4 S-Boxes, s*boxgates* produce results that, as the article will show, can be greatly improved, which is a price for versatility.

3. Setting Objectives

The purpose of our article is to present a method and a utility for generating a bit-sliced description of bijective 4×4 S-Boxes based on a ternary logical instruction, which provides better results compared to existing ones, and this will make it possible to increase the speed and security of hardware and software implementations of a wide range of cryptographic algorithms, which use S-Boxes of a given type.

S-Boxes representation format for bit-sliced implementation

In the specifications of cryptographic algorithms, S-Boxes are mostly specified in the form of LUT tables. For example, the 4×4 S-Box of the PRESENT cipher has the form shown in Table 1

Table 1

LUT-table of S-Box 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

In the bit-sliced representation, LUT tables are treated as logical functions defined by truth tables. For example, the S-Box of the PRESENT cipher will have the form shown in Table 2.

Table 2

Bitsliced-oriented representation of S-Box 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 this S-Box in the form of a truth table will have the following form: S(x) = y, where $x = \{x_0, x_1, x_2, x_3\}$ = {0xff00, 0xf0f0, 0xcccc, 0xaaaa}— input bit-sliced variables, $y = \{y_0, y_1, y_2, y_3\}$ = *{0x0ed9, 0x3687, 0xa74c, 0x659a}*—the output bit-sliced variables defining a specific substitution table, and we will call the 16-bit numbers that specify x and *y as* vectors. The task of finding a bit-sliced S-Box representation according to the BGC criterion can be formulated as follows: given four base vectors *base* = $\{x_0, x_1, x_2, x_3\}$, you need to find the vectors $y = \{y_0, y_1, y_2, y_3\}$ using the minimum number of ternary logical instructions *ternary logic (a, b, c, imm8)*.

4. Preliminary Calculations

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

1. For each 16-bit vector ***v*** there is a BGC(v)—the minimum number of ternary instructions required for its representation, the so-called "complexity" of the vector.

Since vectors are represented as 16-bit numbers, there are 65536 vectors in total, four of which are based vectors *base* = $\{x_0\text{-}x_3\}$ and two are logical constants *const* = {*0x0000, 0xffff*} to denote 0 and 1 for which BGC is 0, so there remain 65530 vectors whose complexity needs to be estimated. Table 3 presents the found distribution of vectors according to their BGC value.

Table 3

Distribution of 16-bit vectors by BGC

BGC	0	1	2	3
Number of vectors	6	936	34250	30344

As can be seen from Table 3, a maximum complexity is 3, meaning that any 16-bit vector can be represented using no more than 3 ternary instructions. This gives an upper estimate of the bit-sliced complexity of an arbitrary S-Box described by four vectors y_0-y_3 equal to 12th TI.

For an indirect preliminary assessment of the "complexity" of the S-Box, such an indicator as the total value of the complexity of the vectors y_0-y_3 can be used: the closer the total value is to 12, the more TI should be expected in the bit-sliced description and vice versa. For example, S-Box UDCIKMP11 (bit-sliced description requires 4 instructions) has the minimum total value of 6 of all S-Boxes considered in the article, and S-Box mCrypton_S0 has the maximum total value of 12 (bit-sliced description requires 8 instructions).

2. Construction of a LUT table for representing all graphs to a depth of ***ge* = *2*** instructions.

The table is built step by step. In the first step, the table q_0 is built, which contains all possible values that can be obtained from the base vectors x_0-x_3 with the help of one ternary instruction and which are not included in *base* and *const*. For this, the *VECT_SQUARE* algorithm is used, which forms all possible combinations from the input vectors using TI.

There are a total of 936 such vectors. The generated values are entered into the table, and the x_0-x_3 values are not stored in the table to save memory, although they are implicitly present for each row. Therefore, the table $\boldsymbol{q_0}$ = *VECT_SQUARE* ($\{x_0, x_1, x_2, x_3\}$) has a dimension of 936×1 (Fig. 2).

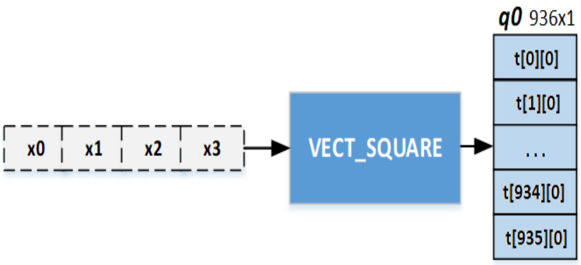

Figure 2: Formation of the table q_0

Each i^{th} line of the table $\boldsymbol{q_0}$ can be considered as a new basis $\{x_0\text{-}x_3, q_0[i]\}$, which is used to generate all possible vectors in the next step, and the LUT lines of the tables themselves will be called graphs.

So, in particular, in the second step, the table $\boldsymbol{q_1}$ = *GEN_TABLE*($\boldsymbol{q_0}$) is generated, for which the *VECT_SQUARE* algorithm again generates from each row of the table $\boldsymbol{q_0}$ all possible values that can be obtained from the base vectors x_0-x_3 and $q_0[i]$ using one ternary instruction, lines are formed from two vectors and added to the general table. After that, logically equivalent graphs are filtered: if the rows of table $\boldsymbol{q_1}$ contain the same values in any order, then only one row remains (Fig. 3).

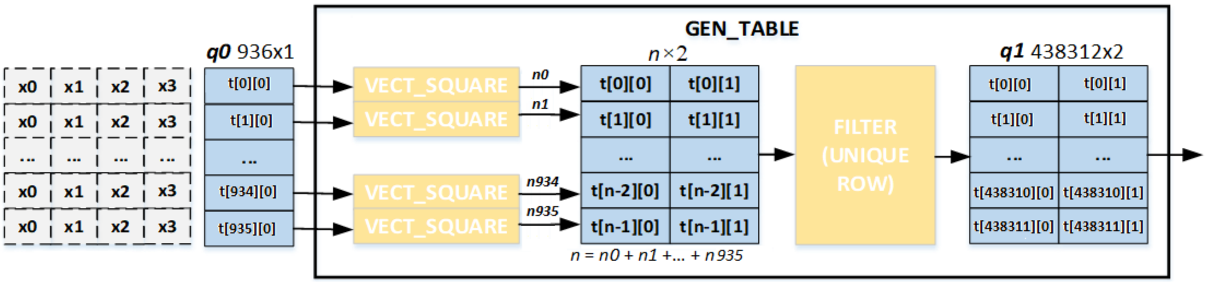

Figure 3: Formation of the table $\mathbf{q_1}$

Table $\boldsymbol{q_1}$ has a dimension of 438312×2, and the branching coefficient when going from table $\boldsymbol{q_0}$ to $\boldsymbol{q_1}$ is equal to 438312/936 = 468.3. The obtained step-by-step results are presented in Table 4.

Table 4

Qualities of LUT-tables q_0-q_1

Table	q_0	q_1
Dimensionality	936×1	438312×2
Branching	-	468.3

Further construction of the tables is impractical, as they require too much memory due to the large value of the branching coefficient. Table $\boldsymbol{q_1}$ will be used to form all possible unique graphs for the representation of vectors y.

5. Search Algorithm of Bitsliced Representation

At the top level of the search algorithm, all values y_0-y_3 are sorted, the matrix of candidate graphs $\boldsymbol{gr_i}$ = *STEP_0*(y_i) is generated for each of them from the precomputed LUT-table q1 and transferred to the depth-first search algorithm *FIND_BS*($\boldsymbol{gr_i}$).

The depth-first search algorithm FIND_BS finds the remaining values y trying to use the minimum TI and returns the constructed complement matrices of graphs $\boldsymbol{gr_0}$-$\boldsymbol{gr_3}$. From the obtained results, the graph with the minimum BGC value is selected (Fig. 4).

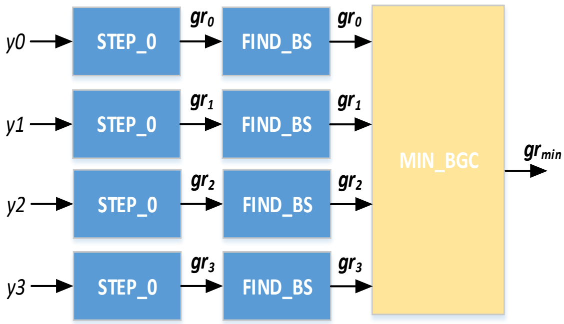

Figure 4: Generalized structure of the S-Box bit-sliced description search algorithm

So, the search algorithm performs four iterations, starting with different values of y. Let us denote this initial value by y_{start}. At the stage $\boldsymbol{gr_i}$ = *STEP_0*(y_{start}), using the LUT table $\boldsymbol{q_1}$, the matrix of graphs $\boldsymbol{gr_i}$, is generated, containing all possible graphs with the vector y_{start} at a certain depth d_{start} of the gates. Depending on which BGC group the y_{start} vector belongs to, heuristically selected d_{start} values are presented in Table 5 to ensure acceptable calculation time and amount of required memory.

Table 5
Depth of generation of graphs containing y_{start} in *STEP_0*

BGC-group y_{start}	1	2	3
d_{start}	2	3	3

So, if, for example, $BGC(y_0) = 2$, then the matrix of graphs $\boldsymbol{gr_0}$ after STEP_0 will contain all possible graphs with a length of 3 instructions ($d_{start} = 3$) in which the vector y_0 occurs.

Next, the candidate graphs in $\boldsymbol{gr_i}$ are sorted into three groups: ***gr_1y***, ***gr_2y***, ***gr_3*** with the same number of vectors *y* in each graph of the group—1, 2, and 3, respectively. We denote this number by *y* _find. Next, the search is conducted for each non-empty group separately according to Fig. 5.

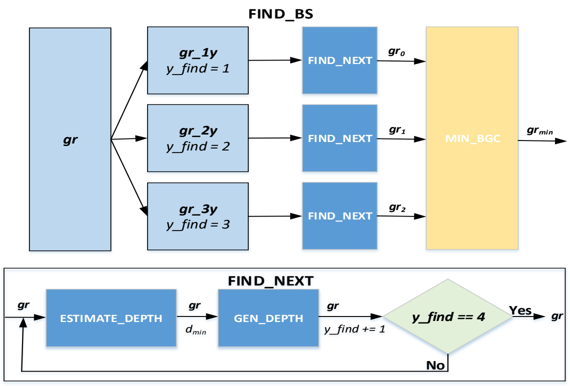

Figure 5: Generalized scheme of searching bit-sliced representation by FIND_BS algorithm

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

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

After the d_{min} estimate is found using the GEN_DEPTH algorithm, a transition is made from the set of graphs with y_find = n_y to the set of graphs with y_find = n_y + 1.

For this, the graphs with the found value d_{min} are selected from the group ***gr*** and a step forward ***gr*** = *GEN_TABLE(**gr**$_{min}$)* is made. For the generated set **gr,** graphs with the found value $d = d_{min}$—*1* are selected again, a step forward is made for them, and so on, until *d* becomes equal to 0. After that, only graphs containing n_y + 1 values of *y* are selected for the group. Then these steps are repeated until all values of *y* are found.

The most computationally intensive procedures ESTIMATE_DEPTH and GEN_DEPTH are implemented by using GPU and OpenCL technology, which makes it possible to significantly parallelize calculations and reduce the algorithm's operating time.

At each step, the FIND_BS algorithm evaluates the minimum distance d_{min}, at which the nearest value of y_x is located, and generates the corresponding graphs. As shown in Fig. 6, this route starts with the graphs containing y_a, generated using STEP_0, from which the nearest value y_b is located at the distance d_{ab} of the 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, not always moving in minimum steps along the trajectory from vector y_a to y_d gives the optimal result in general (although it is so in most cases). There may be a situation when choosing the minimum value of *d* in the first steps leads to larger values of *d* in the following steps and, as a result, to a non-optimal logical representation.

For example, let's assume that at the first step, we got $d_{ab} = 1$, at the second $d_{bc} = 2$, and at the third—$d_{cd} = 2$, that is, the route will be a total of 5 TIs (Fig. 6). But it is possible that if at the first step, we followed a different route and graphs with $d_{ab} = 2$ would be selected, then in the second step it would be possible to find the value of y_c with $d_{bc} = 1$ and in the third step y_d with $d_{cd} = 1$, and we would get a shorter total route with 4 TIs. Hence, the second route resulted in a bit-sliced representation with a lower BGC value.

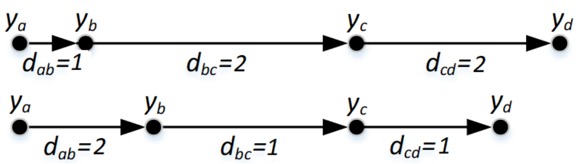

Figure 6: Finding the bit-sliced description for different routes

To take into account different possible routes in the search algorithm, refinement searches are carried out according to the scheme presented in Fig. 7. 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} i $d_{min}+1$, 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}, $d_{min}+1$ i $d_{min}+2$, after which the SEARCH_2Y search is applied to the found graphs with $y\_find = 2$.

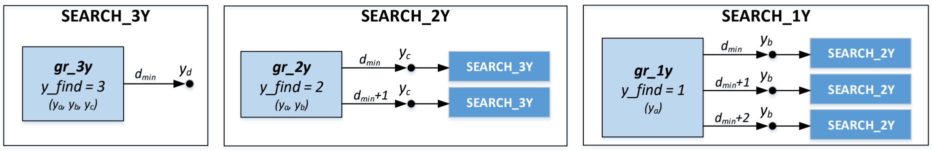

Figure 7: Refinement search scheme in the FIND_BS algorithm

6. Results

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.

To evaluate our algorithm, 225 4×4 S-Boxes of various cryptographic algorithms were taken. The open-source *sboxgates* project was used to obtain a BGC estimate for selected S-Boxes and to be able to compare with our results. Bitsliced descriptions of S-Boxes obtained by our method are available at the link [15].

The results are presented in Table 6. Column data in Table 6 should be interpreted as follows:

LUT is a representation of S-Box in the tabular form, 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.

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

CY is BGC of vectors y_0-y_3. The line '2133' should be interpreted as follows: BGC(y_0) = 2, BGC(y_1) = 1, BGC(y_2) = 3, BGC(y_3) = 3.

OURS contains BGC values obtained using the method described in the article.

SG contains the BGC value obtained using the *sboxgates* utility; the number of iterations for the search was set to 1000 [9]. The red color in the SG column indicates the S-Boxes that have a higher BGC value compared to the one obtained by our algorithm. The yellow ones have the same BGC value as our algorithm.

Table 6
BGC comparison for different S-Boxes

S-Box	LUT	BS	CY	OURS	SG
Piccolo	e4b238091a7f6c5d	aaa5_fc03_1e1d_cd94	1123	4	4
Piccolo^{-1}	68341eca5792df0b	b4e2_3369_aaa5_b714	3213	4	4
LAC	e9f0d4ab128376c5	44d7_f035_3ac5_9996	2222	5	6
Prost	048f15e927acbd63	3ccc_6a6a_d748_b2b8	1132	4	4
Rectangle	65ca1e79b03d8f42	39ac_6867_a569_2dd2	3222	6	9
Rectangle^{-1}	94fae106c7382b5d	a91d_c396_369c_e625	3223	6	8
Minalpher	b34128cf5de069a7	66e1_97c4_d493_a38b	2333	8	9
Skinny	c6901a2b385d4e7f	aaa5_fc03_e1e2_cd94	1123	4	4
TWINE	c0fa2b9583d71e64	256d_ec85_6a3c_1ee4	3322	7	8
PRINCE	bf32ac916780e5d4	5473_f322_131f_62c7	2223	7	8
Lucifer_S0	cf7aedb026319458	907b_6237_075e_5c66	3233	7	8
Lucifer_S1	72e93b04cd1a6f85	6b2c_b385_3837_a639	3323	8	9
Present	c56b90ad3ef84712	0ed9_3687_a74c_659a	3332	7	7
Present^{-1}	5ef8c12db463079a	c19e_2697_ad46_69a5	3332	7	9
JH_S0	904bdc3f1a26758e	c2b9_b8b4_9ec8_31d9	3233	7	8
JH_S1	3c6d5719f204bae8	f18a_493e_7325_11f9	3332	8	9
Iceberg_S0	d7329ac1f45e60b8	c971_1f43_592e_4597	3223	7	8
Iceberg_S1	4afc0d9be6173582	41ee_2b2d_9b86_3ce4	2333	7	8

Luffa	de015a76b39cf824	3d23_98d3_53e2_1759	3333	6	8
Noekeon	7a2c48f0591e3db6	6a6a_a959_d847_7741	1232	6	6
Hummingbird_1_S1	865f1ca9eb2470d3	43e9_592e_974a_d29c	2233	7	9
Hummingbird_1_S2	07e15b823ad6fc49	b664_7c16_1ba6_953a	3332	8	9
Hummingbird_1_S3	2ef5c19ab468073d	89d6_a61e_6587_e16c	3333	8	9
Hummingbird_1_S4	0734c1afde6b2895	6bd0_879a_1ec6_c9a6	3333	8	9
Hummingbird_1_S1^{-1}	d4afb21c07695e83	689d_368b_a63c_9a59	3322	7	9
Hummingbird_1_S2^{-1}	0378e4b16f95da2c	b658_9b34_6356_1ec6	3333	8	9
Hummingbird_1_S3^{-1}	c50e93adb6784f12	29d9_368b_a768_65b2	2333	8	8
Hummingbird_1_S4^{-1}	05c23fa1de6b4897	6b64_9726_8e78_c9b2	2333	8	9
Hummingbird_2_S1	7ce9215fb6d048a3	658e_16c7_c395_85e9	3323	8	9
Hummingbird_2_S2	4a168f7c30ed59b2	6cb2_1ce9_c56a_7964	3323	8	9
Hummingbird_2_S3	2fc156ade8340b97	63c6_89b6_a563_e49a	2323	7	9
Hummingbird_2_S4	f4589721a30e6cdb	e919_7827_9b61_c2b5	2233	7	9
Hummingbird_2_S1^{-1}	b54fc690d3e81a27	2d59_853e_e629_934b	3333	8	9
Hummingbird_2_S2^{-1}	92f80c364d1e7ba5	6a2d_9ba4_78c6_b645	3333	8	9
Hummingbird_2_S3^{-1}	c30ab45f9e6d2781	4b99_2ee1_369a_a9d2	2233	7	8
Hummingbird_2_S4^{-1}	a76912c5348fdeb0	7c49_3ac6_6927_599a	3232	7	9
DES_S1_0	e4d12fb83a6c5907	2ae5_9c27_8771_b16c	3232	7	8
DES_S1_1	0f74e2d1a6cb9538	9d52_265e_4b36_78c6	3333	7	8
DES_S1_2	41e8d62bfc973a50	279c_4b35_39e4_5d92	2323	7	8
DES_S1_3	fc8249175b3ea06d	9a27_c993_5e89_87e1	3232	8	9
DES_S2_0	f18e6b34972dc05a	992d_5a99_8679_4b63	2223	7	8
DES_S2_1	3d47f28ec01a69b5	69d2_919e_58b9_e41b	2232	7	9
DES_S2_2	0e7ba4d158c6932f	965a_8d66_e81e_b1cc	2332	7	8
DES_S2_3	d8a13f42b67c05e9	c927_6e61_47b4_a539	2222	7	9
DES_S3_0	a09e63f51dc7b428	964d_2ed8_5879_1be4	3332	8	9
DES_S3_1	d709346a285ecbf1	7a89_5c63_69d2_e41b	3222	7	9
DES_S3_2	d6498f30b12c5ae7	6939_d827_e562_9369	2232	7	8
DES_S3_3	1ad069874fe3b52c	9666_a794_5e92_3aa5	2223	7	9
DES_S4_0	7de3069a1285bc4f	b4c6_e827_92ad_994b	3332	8	9
DES_S4_1	d8b56f03472c1ae9	e827_4b39_66b4_92ad	3323	8	9
DES_S4_2	a690cb7df13e5284	49b5_99d2_2d63_17e4	3233	8	9
DES_S4_3	3f06a1d8945bc72e	99d2_b64a_e81b_2d63	2333	8	9
DES_S5_0	2c417ab6853fd0e9	d962_5a96_4cf1_9e58	3233	7	8
DES_S5_1	eb2c47d150fa3986	6c4b_8579_9c27_35e2	3323	7	9
DES_S5_2	421bad78f9c5630e	87b8_9d61_b15a_2b6c	2333	8	9
DES_S5_3	b8c71e2d6f09a453	1aa7_63ac_9369_ca99	3223	8	9
DES_S6_0	c1af92680d34e75b	929d_7a49_b46c_e61a	2233	7	9
DES_S6_1	af427c9561de0b38	ac63_0db6_691b_66d2	2332	7	9
DES_S6_2	9ef528c3704a1db6	6867_a54e_c996_718d	2323	7	9
DES_S6_3	432c95fabe17608d	c3d8_9a69_1bc6_8d72	3222	7	9
DES_S7_0	4b2ef08d3c975a61	26da_5a99_691e_9d92	3222	7	9
DES_S7_1	d0b7491ae35c2f86	69a5_ad19_b38c_266d	2323	7	8
DES_S7_2	14bdc37eaf680592	4b9c_26da_87e4_626d	3332	8	9
DES_S7_3	6bd814a7950fe23c	994e_9aa5_78c3_4b96	3222	7	9
DES_S8_0	d2846fb1a93e50c7	4b65_d839_8d72_96e1	3322	7	9
DES_S8_1	1fd8a374c56b0e92	691e_27c6_ac72_4a67	2333	7	8
DES_S8_2	7b419ce206adf358	9c72_5a65_36c3_781b	2223	7	9
DES_S8_3	21e74a8dfc90356b	87e4_639c_d12d_b58a	3223	7	9
Serpent_S0	38f1a65bed42709c	c396_9764_19b5_52cd	2333	7	8
Serpent_S1	fc27905a1be86d34	2e93_b44b_568d_6359	3233	7	8
Serpent_S2	86793cafd1e40b52	25e9_4da6_a4d6_639c	2332	7	9
Serpent_S3	0fb8c963d124a75e	913e_e952_b4c6_63a6	3333	8	9
Serpent_S4	1f83c0b6254a9e7d	b856_e692_69ca_d24b	2332	7	9
Serpent_S5	f52b4a9c03e8d671	1ce9_7493_662d_d24b	3322	7	9

Serpent_S6	72c5846be91fd3a0	5b94_196d_69c3_3e89	3323	7	9
Serpent_S7	1df0e82b74ca9356	1cb6_c716_a9d4_7187	2333	8	9
Serpent_$S0^{-1}$	d3b0a65c1e47f982	7295_1ee1_9a36_3947	3233	7	8
Serpent_$S1^{-1}$	582ef6c3b4791da0	695a_2679_45bc_3d91	2333	7	8
Serpent_$S2^{-1}$	c9f4be12036d58a7	6837_9c2d_c6b4_9a56	3332	7	8
Serpent_$S3^{-1}$	09a7be6d35c248f1	64b6_56e8_497c_c39a	3332	8	9
Serpent_$S4^{-1}$	5083a97e2cb64fd1	66b4_7ac1_2dd8_e469	2323	7	9
Serpent_$S5^{-1}$	8f2941deb6537ca0	61cb_36d2_5b86_1d6a	2332	7	9
Serpent_$S6^{-1}$	fa1d536049e72c8b	e60b_2d59_9c63_8a3d	3323	7	9
Serpent_$S7^{-1}$	306d9ef85cb7a142	16f8_4b6c_9c65_2d59	3333	8	9
GOST_1	4a92d80e6b1c7f53	2ab6_7991_b38a_f614	3333	8	8
GOST_2	eb4c6dfa23810759	84eb_607d_23d3_ea62	3322	7	8
GOST_3	581da342efc7609b	c71a_1f49_9bb0_ca2d	3333	8	9
GOST_4	7da1089fe46cb253	19e6_4f83_b585_d0cb	2223	7	8
GOST_5	6c715fd84a9e03b2	4ee2_0977_ea25_647c	3233	8	9
GOST_6	4ba0721d36859cfe	f486_ea91_c336_59d2	3323	8	8
GOST_7	db413f590ae7682c	a6a3_9c65_5e32_08fb	2332	7	8
GOST_8	1fd057a4923e6b8c	e946_98b6_3e62_2537	3332	7	9
LBlock_S0	e9f0d4ab128376c5	44d7_f035_3ac5_9996	2222	5	6
LBlock_S1	4be9fd0a7c562813	22be_0f35_9996_c53a	2222	5	7
LBlock_S2	1e7cfd06b593248a	c53a_22be_9996_0f35	2222	5	7
LBlock_S3	768b0f3e9acd5241	0fac_5ca3_22eb_9969	2222	5	7
LBlock_S4	e5f072cd1849ba63	3ac5_44d7_f035_9996	2222	5	6
LBlock_S5	2dbcfe097a631845	22be_c53a_0f35_9996	2222	5	7
LBlock_S6	b94e0fad6c573812	22eb_0fac_9969_5ca3	2222	5	7
LBlock_S7	daf0e49b218375c6	44d7_f035_9996_3ac5	2222	5	6
LBlock_S8	87e5fd06bc9a2413	0f35_22be_9996_c53a	2222	5	7
LBlock_S9	b5f0729d481cea36	3ac5_9996_f035_44d7	2222	5	6
SC2000_4	25ac7f1bd609483e	a9ac_933a_c2b5_49f2	2333	8	8
MIBS	4f38dac0b57e2619	897a_2e53_3d26_c716	3333	8	9
KLEIN	74a91fb0c3268ed5	716c_e923_2e65_c279	3333	8	9
Panda	0132fc9ba6875ed4	65f0_fa30_2b9c_58d6	2233	7	8
MANTIS	cad3ebf789150246	0377_c8d5_a0fa_0eec	2212	6	8
GIFT	1a4c6f392db7508e	c6aa_9a3c_8d72_1ee1	2222	6	6
UDCIKMP11	086d5f7c4e2391ba	d2aa_03fc_ce64_7878	2121	4	4
Luffa_v1	7dbac4835f60912e	925e_8733_c68d_3387	2232	7	9
Enocoro_S4	139a5e72d0cf486b	ad2c_5d70_c8ea_8957	3323	7	9
Qarma_sigma0	0e2a9f8b6437dc15	30fa_bb22_0dae_dcb0	2123	7	8
Qarma_sigma1	ade6f735980cb124	1b17_88be_507d_31f2	2222	7	9
Qarma_sigma2	b68fc09e3745d21a	90dd_1e9a_a38b_5b49	2333	7	8
Midori_Sb0	cad3ebf789150246	0377_c8d5_a0fa_0eec	2212	6	8
Midori_Sb1	1053e2f7da9bc846	3f50_d1d4_8af8_0dcd	2222	7	8
Anubis_S0	d7329ac1f45e60b8	c971_1f43_592e_4597	3223	7	8
Anubis_S1	4afc0d9be6173582	41ee_2b2d_9b86_3ce4	2333	7	8
Khazad_P	3fe054bcda967821	27c6_19b6_5a47_9553	3333	8	9
Khazad_Q	9e56a23cf04d7b18	a993_1d8e_317a_7945	3333	8	9
Fox_S1	2519eac8647fdb03	38f8_1f52_ad31_bc0e	2333	7	8
Fox_S2	b41f03eda875c296	53c9_9cca_a569_4cad	3323	6	8
Fox_S3	dab14389572cf06e	98c7_db11_d626_13ad	3323	8	9
Whirlpool_E	1b9cd6f3e874a250	135e_4d78_35e2_44d7	3332	8	9
Whirlpool_R	7cbde49f638a2510	0cde_21bb_1b95_62cd	2233	7	9
SMASH_256_S1	6dc7f13a8b5024e9	c396_641f_52d9_867a	2333	7	8
SMASH_256_S2	1b60ed5ac29738f4	65b2_c974_5a96_5c63	3322	7	9
SMASH_256_S3	429c81e7f50b6a3d	a95c_93c9_79c2_cba4	2233	7	9
CS_cipher_G	a602be18d453fc79	b1b1_7722_583b_dd50	1132	6	6
GOST2_1	6af43850de712bc9	e326_474d_3617_ad54	3233	8	9

GOST2_2	e0817a56d2493fcb	e925_65d1_b2b1_b958	2323	8	9
Magma_1	c462a5b9e8d703f1	47d1_4d27_695c_ece0	3332	7	9
Magma_2	68239a5c1e47bd0f	b2b2_aec1_9a2d_b958	1333	7	8
Magma_3	b3582fade174c960	31e9_5da4_4573_26a7	3333	8	9
Magma_4	c821d4f670a53e9b	e453_29f1_b5c4_d958	3333	8	9
Magma_5	7f5a816d093eb42c	9a9a_a8c7_5c4b_16a7	1333	7	8
Magma_6	5df692cab78143e0	45d6_524f_63ac_2b17	3322	7	8
Magma_7	8e25691cf4b0da37	35a3_939a_e516_d568	3333	8	9
Magma_8	17ed05834fa69cb2	764c_2b2e_ce86_52ab	3223	8	8
CLEFIA_SS0	e6ca872fb14059d3	619d_54a7_81eb_f3a0	3332	7	8
CLEFIA_SS1	640d2ba39cef8751	1f68_6e0b_2cf1_e9a8	3332	7	9
CLEFIA_SS2	b85ea64cf72310d9	c19b_43ec_0f39_db05	3323	7	9
CLEFIA_SS3	a26d345e0789bfc1	7c89_62ec_3297_ba58	3333	7	8
Golden_S0	035869c7dae41fb2	6768_2dd4_e692_71a6	2333	8	9
Golden_S1	03586cb79eadf214	1f68_9ab4_36d2_59c6	3333	8	9
Golden_S2	03586af4ed9217cb	c768_63d4_a972_b646	3332	8	9
Golden_S3	03586cb7a49ef12d	9d68_9ab4_59d2_b4c6	3333	8	9
Twofish_Q0_T0	817d6f320b59eca4	7a29_b43c_52f4_0e6e	3232	7	8
Twofish_Q0_T1	ecb81235f4a6709d	c50f_9b83_1d65_d1d4	2332	7	9
Twofish_Q0_T2	ba5e6d90c8f32471	076b_653c_5c1b_cc65	3232	7	8
Twofish_Q0_T3	d7f4126e9b3085ca	d385_60cf_86e6_2717	3232	7	8
Twofish_Q1_T0	28bdf76e31940ac5	649e_c8f8_21f5_873c	3222	7	9
Twofish_Q1_T1	1e2b4c376da5f908	b62a_1bb2_15ce_3ac9	3332	7	8
Twofish_Q1_T2	4c75169a0ed82b3f	aec2_862f_f2a4_e45c	3333	7	9
Twofish_Q1_T3	b951c3de647f208a	c8d3_0fd4_9da1_0c6f	3232	7	9
Serpent_type_S0	03567abcd4e9812f	9de0_879c_c47a_a956	3332	7	7
Serpent_type_S1	035869a7bce21fd4	6768_e694_2dd2_71a6	2323	7	9
Serpent_type_S2	035869b2d4e1af7c	b568_e714_74d2_6966	3332	6	8
Serpent_type_S3	03586af4ed9217cb	c768_63d4_a972_b646	3332	8	9
Serpent_type_S4	03586cb79eadf214	1f68_9ab4_36d2_59c6	3333	8	9
Serpent_type_S5	03586cb7a49ef12d	9d68_9ab4_59d2_b4c6	3333	8	9
Serpent_type_S6	03586cb7ad9ef124	1f68_9ab4_59d2_36c6	3332	7	8
Serpent_type_S7	03586cb7dae41f29	a768_2db4_66d2_b1c6	3222	7	8
Serpent_type_S8	03586cf1a49edb27	3d68_9a74_e952_b4c6	2233	7	9
Serpent_type_S9	03586cf2e9b7da41	3768_5974_2dd2_9e46	3323	7	9
Serpent_type_S10	03586df19c2ba74e	9b68_e274_bc52_29e6	3333	7	9
Serpent_type_S11	03586df274eba19c	dc68_8774_1dd2_6966	3222	7	9
Serpent_type_S12	03586df2c9a4be17	3768_a974_b4d2_d266	3222	7	8
Serpent_type_S13	03586fa179e4bcd2	7668_6d34_9572_53a6	3332	7	9
Serpent_type_S14	0358749ef62badc1	79c8_63b4_1f92_a956	3332	7	9
Serpent_type_S15	035879beadf4c261	17e8_5e94_65d2_8676	2332	7	9
Serpent_type_S16	03589ce7adf46b12	2778_1ee4_b5c2_6696	2232	7	7
Serpent_type_S17	0358ad94f621cb7e	b178_d3a4_e712_6966	3332	6	8
Serpent_type_S18	0358bc6fe9274ad1	63b8_59e4_2dd2_ca96	3323	7	8
Serpent_type_S19	035a7cb6d429e18f	d968_93b4_94da_a956	3332	7	9
BLAKE_1	ea489fd61c02b753	127b_62e5_b8a3_f170	2332	7	8
BLAKE_2	b8c052fdae367194	43c7_9ad4_1f61_74d1	2332	7	9
BLAKE_3	7931dcbe265a40f8	c8f2_56b1_4bc5_445f	2332	7	9
BLAKE_4	905724afe1bc683d	adc1_99ac_55d8_c68d	3323	7	8
BLAKE_5	2c6a0b834d75fe19	b26a_3f06_34ad_dea0	3232	7	9
BLAKE_6	c51fed4a0763928b	d0b9_067b_ae98_9a2e	3333	7	9
BLAKE_7	db7ec13950f4862a	949b_2d1d_e44e_05e7	2223	7	7
BLAKE_8	6fe9b308c2d714a5	459e_ad07_4a37_9c3a	3233	7	9
BLAKE_9	a2847615fb9e3cd0	6f05_69b8_1b33_57d0	2323	7	9
GOST_IETF_1	96328b17a4efc0d5	5d31_de82_0dae_c8e5	3323	8	9
GOST_IETF_2	37e98af0526cb4d1	587c_6d46_1667_d14b	3333	7	9

GOST_IETF_3	e462b3d8cf5a0719	8bd1_2747_2a3d_e670	3333	8	9
GOST_IETF_4	e7acd13902b4f856	349d_d81b_9647_54f2	3333	8	9
GOST_IETF_5	b5198df0e423c7a6	5179_b362_ed41_286f	2233	7	9
GOST_IETF_6	3adc120b75948fe6	748e_eb0c_e1a3_2795	3233	7	8
GOST_IETF_7	1d297a608c45f3be	d32a_9e52_f074_781b	3223	7	8
GOST_IETF_8	baf50ce8623917d4	48e7_e16c_2747_7c0d	3333	8	9
Kuznyechik_nu0	253b69ea04f18dc7	74e8_e652_84dd_ac2e	3323	8	8
Kuznyechik_nu1	76c90f8145bed23a	9c6c_1b27_ec23_56a9	2222	5	5
Kuznyechik_sigma	cd048bae3952f167	12f3_d48b_d9e0_b722	2332	7	8
Optimal_S0	012d47f68bc93ea5	6f48_a4f8_72e4_9a6a	3232	7	8
Optimal_S1	012d47f68be359ac	e748_94f8_4ee4_3a6a	3322	6	7
Optimal_S2	012d47f68be3ac59	b748_64f8_1ee4_ca6a	2322	7	8
Optimal_S3	012d47f68c53aeb9	f348_26f8_78e4_cc6a	2332	7	8
Optimal_S4	012d47f68c9bae53	3f48_62f8_b8e4_cc6a	2332	7	8
Optimal_S5	012d47f68cb9ae35	3f48_a2f8_74e4_cc6a	2232	7	8
Optimal_S6	012d47f68cb9ae53	3f48_62f8_b4e4_cc6a	2322	7	7
Optimal_S7	012d47f68ceba935	3f48_86f8_5ce4_e86a	2333	8	8
Optimal_S8	012d47f68e95ab3c	b748_8af8_72e4_6c6a	2233	6	7
Optimal_S9	012d47f68eb359ac	e748_92f8_4ee4_3c6a	3322	7	8
Optimal_S10	012d47f68eb5a93c	b748_8af8_56e4_6c6a	2233	7	8
Optimal_S11	012d47f68eba59c3	6f48_52f8_8ee4_b46a	3233	7	9
Optimal_S12	012d47f68eba93c5	5f48_c2f8_2ee4_b46a	2233	7	9
Optimal_S13	012d47f68ec95ba3	6f48_16f8_e2e4_b86a	3333	7	9
Optimal_S14	012d47f68ecb395a	af48_46f8_9ae4_786a	2333	8	9
Optimal_S15	012d47f68ecb93a5	5f48_86f8_6ae4_b86a	2333	8	9
Num1_DL_04_0	0bc5619a3ef8d427	1ec6_b61c_c792_956a	3222	7	7
Num1_DL_04_1	0cda5be7f6213894	616e_83d6_17e8_59b4	2223	7	8
Num1_DL_13_0	0c9761f23b4ed8a5	7a46_9c5a_4bd8_936c	3232	7	9
Num1_DL_13_1	0c97f2613b4ea5d8	da16_6c5a_1b78_639c	2232	7	9
Num1_DL_13_2	0b85fc36e47921da	c936_47b8_95d2_6c5a	2232	7	8
Num1_DL_13_3	0d4b7e926a3581fc	d26a_c936_47b8_6c5a	3222	7	8
Num1_DL_22_0	0d82eb75f63c419a	c936_1bd2_8778_65e2	2323	6	8
Num1_DL_22_1	0be1a7d46c9f5832	2e56_1be4_c936_5c6a	3223	6	8
Num1_DL_22_2	0b69c53ed7842af1	659a_4bb4_72c6_c36a	2232	6	8
Num1_DL_22_3	0e95f8a73b6c41d2	4a76_5c9a_87d2_639c	3322	7	9
mCrypton_S0	4f38dac0b57e2619	897a_2e53_3d26_c716	3333	8	9
mCrypton_S1	1c7a6d53fb20849e	d32a_a176_879c_43e5	3333	8	9
mCrypton_S2	7ec209da3f5864b1	4ae6_3647_538b_c761	3333	8	9
mCrypton_S3	b0a7d642ce3915f8	cb15_6378_46ad_7c19	3333	8	9
				Σ=1582	Σ=1867

In general, as evidenced by the results in Table 7, the method we developed showed significantly better results compared to the competitor represented by the *sboxgates* utility. For 203 S-Boxes out of 225 (90.2%), our method provides a bit-sliced description with fewer TIs. The total number of ternary instructions to represent all 225 S-Boxes in our method is 1582, which is 15.3% less compared to 1867 instructions for the *sboxgates* utility. The *sboxgates* utility did not generate a bit-sliced description with fewer instructions than obtained by our method for any S-Box, and for only 22 S-Boxes (9.8%) it was able to generate a bit-sliced description with the same BGC value as our algorithm.

7. Conclusions

The paper presents a method for generating a bit-sliced description of arbitrary 4×4 bijective S-Boxes with a reduced number of ternary logic instructions. The obtained descriptions make it possible to generally increase the speed of software implementations of the corresponding crypto-algorithms on any processors that support the 3-operand ternary logic instruction

(CPU/GPU). To date, the method proposed in the article is the most effective method known to us according to the BGC criterion, which confirms 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 3 gates using GPU, IDDFS algorithm for searching and cutting options, refinement search, which by this set of measures ensure its efficiency and acceptable speed of action.

8. References

[1] A. Bessalov, et al., Computing of Odd Degree Isogenies on Supersingular Twisted Edwards Curves, in: Workshop on Cybersecurity Providing in Information and Telecommunication Systems, vol. 2923 (2021) 1–11.

[2] A. Bessalov, V. Sokolov, P. Skladannyi, Modeling of 3- and 5-Isogenies of Supersingular Edwards Curves, in: 2nd International Workshop on Modern Machine Learning Technologies and Data Science, no. I, vol. 2631 (2020) 30–39.

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

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

[5] 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.

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

[7] J. Zhang, M. Ma, P. Wang, Fast Implementation for SM4 Cipher Algorithm Based On Bit-Slice Technology, International Conference on Smart Computing and Communication, 2018, 104–113.

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

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

[10] 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.

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

[12] K. Stoffelen, Optimizing S-Box Implementations for Several Criteria Using SAT Solvers, Proc. 23rd International Conf. on Fast Software Encryption, 2016, 140–160.

[13] 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.

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

[15] Z. Bao, et. al., nPeigen—a Platform for Evaluation, Implementation, and Generation of S-boxes, IACR Transactions on Symmetric Cryptology, 2019, 330–394.

[16] D. Mercadier, Usuba, Optimizing Bitslicing Compiler, PhD Thesis, Sorbonne University, France, 2020, 195.

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

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

[19] AQ. Sahun, et. al., Devising a Method for Improving Crypto Resistance of the Symmetric Block Cryptosystem RC5 Using Nonlinear Shift Functions. Eastern-European J. of Enterprise Technologies, 5(9) (113) (2021) 17–29. doi: 10.15587/1729-4061.2021.240344