On Black-box optimization in Divide-and-Conquer SAT solving

O. S. Zaikin and S. E. Kochemazov

Matrosov Institute for System Dynamics and Control Theory SB RAS, Irkutsk, Russia

ARTICLE HISTORY
Compiled August 14, 2019

ABSTRACT
Solving hard instances of the Boolean satisfiability problem (SAT) in practice is an interestingly nontrivial area. The heuristic nature of SAT solvers makes it impossible to know in advance how long it will take to solve any particular SAT instance. One way of coping with this disadvantage is the Divide-and-Conquer approach when an original SAT instance is decomposed into a set of simpler subproblems. However, the way it is decomposed plays a crucial role in the resulting effectiveness of solving. In the present study, we reduce the problem of choosing a proper decomposition to a stochastic pseudo-Boolean black-box optimization problem. Several optimization algorithms of different types were used to analyze a number of hard SAT-based optimization problems, related to SAT-based cryptanalysis of state-of-the-art stream ciphers. A meticulous computational study showed that some of the considered optimization algorithms perform much better than the others in the context of the problems from the considered class. It turned out, that the obtained results also pose some cryptographic interest.

KEYWORDS
pseudo-Boolean optimization; black-box optimization; Monte-Carlo method; SAT; divide-and-conquer; cryptanalysis

1. Introduction

Black-box optimization methods have been intensively developing in the last few decades. Such methods operate with objective functions for which the analytic form is unknown [2]. In particular, it means that no gradient information can be obtained for them. Nevertheless, numerous hard problems can be effectively solved by black-box optimization algorithms. In the present paper, we study the applications of black-box optimization methods to solving hard instances of the Boolean satisfiability problem (SAT) [6].

SAT is one of the most well-studied problems in computer science. Despite the fact that it is an NP-complete problem [22], in the last two decades the effectiveness of heuristic SAT solving algorithms has significantly increased. As a result, a number of problems from different areas, say, hardware verification, model checking and bioinformatics, have been effectively reduced to SAT and solved by SAT solvers [6]. One of the areas where SAT solvers allow to obtain quite interesting results is cryptanalysis. The corresponding approach is called *SAT-based cryptanalysis* [43]. We consider SAT-based cryptanalysis instances in the context of the Divide-and-Conquer SAT solving

CONTACT O. S. Zaikin. Email: zaikin.icc@gmail.com

approach, which implies that an original SAT instance is split into disjoint simpler subproblems that can be solved independently (e.g. in parallel).

In [20, 45, 54, 58] a Divide-and-Conquer method was proposed that consists in choosing a set of variables in a SAT instance and varying all possible assignments of their values. Each simplified instance, obtained by substituting the corresponding assignments, is then solved by a SAT solver without any time limit. Such a set of variables can be viewed as an input of a black-box function computing a Monte Carlo-based estimation of the runtime required for solving an original problem. It is clear that the minimum of such a function corresponds to a set of variables that yields the smallest runtime estimation. This function was minimized in [58] by a random search algorithm. In [53], it was minimized by a tabu search algorithm and simulated annealing algorithm. In [55], a modified SAT-based Divide-and-Conquer method was proposed, in which a time limit for solving simplified instances is used. The corresponding modified Monte Carlo-based function was minimized by an evolutionary algorithm and a greedy algorithm in [49, 50]. In the present paper, we study only the first mentioned method, i.e. the one proposed in [20, 45, 54, 58]. Note, that both mentioned functions were studied in application to SAT-based cryptanalysis. In particular, they were minimized by black-box optimization algorithms over the space of all possible subsets of variables encoding the secret key of the considered cipher.

The studies performed in [53, 58] have the following drawbacks: (i) the corresponding objective function was not described properly from the optimization point of view; (ii) no meticulous study on which black-box optimization algorithms suit better for the objective function was performed; (iii) no guide was proposed on how one can use the source code of the function to minimize it on the corresponding problems using other optimization algorithms. Apparently because of these drawbacks, the optimization community is not aware of these hard SAT-related optimization problems. One of the goals of the present paper is to fill this gap. The mentioned objective function is described in detail. An improved objective function of the same type is proposed instead. The proposed function is minimized by several optimization algorithms in application to four hard optimization problems. These four problems, in turn, consist in finding good runtime estimations for SAT-based cryptanalysis of four state-of-the-art stream ciphers.

Thus, the main contributions of the paper are as follows.

(1) A new stochastic pseudo-Boolean black-box objective function for Divide-and-Conquer SAT solving is proposed.
(2) Several pseudo-Boolean black-box optimization algorithms of different types are programmatically implemented to study the proposed function.
(3) The experimental evaluation of the considered optimization algorithms is performed on four hard SAT instances corresponding to cryptanalysis of state-of-the-art stream ciphers.
(4) As a result of the computational study, the best known estimations for SAT-based cryptanalysis of all considered stream cipher are obtained using the proposed function.
(5) The guide on how to use the source code of the proposed objective function in application to the considered optimization problems is presented.

This paper is organized as follows. In the next section, the preliminaries are given. In Section 3, a new stochastic pseudo-Boolean black-box objective function is proposed, that is aimed at finding decompositions with small runtime estimations for hard SAT instances. Section 4 describes optimization algorithms that are further em-

ployed to optimize the proposed objective function. Section 5 contains information on the considered hard optimization problems. In Section 6, the results of computational experiments are presented. In the last sections, the results are discussed and conclusions are drawn.

2. Preliminaries

A Boolean variable x is a variable that can take only two values $x \in \{\text{False}, \text{True}\}$, often represented by $\{0, 1\}$ respectively. A *literal* is either a Boolean variable or its negation $\neg x$. A sequence of literals connected by logical "or", e.g. $x_1 \vee x_2 \vee \neg x_3$, is called a disjunction or a *clause*. It takes the value of True if and only if any of the literals takes this value. A conjunction (logical "and") of clauses is called a Conjunctive Normal Form (CNF). Any Boolean formula can be represented in CNF [59]. The Boolean satisfiability problem (SAT) in its decision variant is then formulated as follows: for a CNF C over Boolean variables from set $X = \{x_1, \ldots, x_n\}$, $|X| = n$ to answer the question whether there exists such an assignment $\alpha = (\alpha_1, \ldots, \alpha_n)$ of variables from X that once each variable x_i is set to α_i, the CNF C becomes True. If such an assignment exists, then it is called *satisfying assignment* and C is called *satisfiable*. If there are no assignments satisfying the formula, then the formula is called *unsatisfiable*.

SAT is the historically first NP-complete problem [14]. It means that it is possible to effectively formulate the majority of combinatorial problems, which arise in practice, in SAT form. Of course, a SAT formulation itself does not make the problem easier to tackle. However, the progress achieved in heuristic modifications of complete SAT solving algorithms in the recent two decades makes it possible to use them today for solving a variety of very difficult combinatorial problems from such areas as software verification, bioinformatics, cryptography, etc. [6]. Since technical progress leads to the ever-more complex programs and cryptographic constructions, approaches for solving SAT in parallel are of particular interest and importance. One such approach is called *Divide-and-Conquer* and implies splitting an original SAT instance into a number of simpler subproblems to be processed in parallel. There exist various Divide-and-Conquer schemes, such as guiding path [65], scattering [35], Cube-and-Conquer [30], and plain partitioning [36]. According to the approach from [20, 45, 53, 54, 58], all possible values of a subset of variables of a given CNF are varied. In the present paper, we study and improve this very approach. Note, that it is a special case of the plain partitioning.

Our goal is to apply the Monte Carlo apparatus to estimating the time required to solve all subproblems constructed by splitting an original SAT instance. In order to do this, certain conditions should be satisfied. For example, the hardness of the subproblems should vary in reasonable limits in order for the Monte Carlo method to work. Hereinafter, we decompose the SAT instances as follows. Assume that C is a Boolean formula in CNF over a set X of Boolean variables. Now suppose that a subset S is specified such that $S \subseteq X$, $|S| = k$, $k \leq n$. The general idea is that we process all instantiations of variables from S by substituting their values into C, and solve the simplified problems. By assigning values $\alpha = (\alpha_1, \ldots, \alpha_k)$ to variables forming S we obtain a simplified formula denoted as $C[S/\alpha]$. Let us refer to a set of simplified formulas produced by assigning all possible different combinations of values to variables from S in C as to *decomposition* of C. Hereinafter we denote it as $D_S[C] = \{C[S/\alpha], \alpha \in \{0,1\}^{|S|}\}$. It is easy to see that $|D_S[C]| = 2^k$. The decomposition is formed by formulas that differ from each other only in values of

variables from S. Then it is natural to assume that they all are weakened more or less similarly compared to the original instance. In fact, this assumption may not hold true for an arbitrary Boolean formula in CNF, but it holds for a wide classes of problems, e.g. the ones arising in cryptanalysis. We will cover this question in more detail below. The very important feature of the plain partitioning approach is that the instances that comprise a decomposition can be processed independently in parallel. If an original formula is unsatisfiable, then all the problems from a decomposition will have to be solved, otherwise as soon as a satisfying assignment is found for at least one simplified problem, it means that we successfully constructed the solution for the original problem.

The technique used to construct a decomposition allows one to use the Monte Carlo method [47] to estimate the time required to solve all problems from a decomposition by solving a small sample of them and extrapolating the result to the whole. While this runtime estimation is very helpful by itself, an important fact is that finding a set S to decompose a problem that yields a reasonably small runtime estimation is a very hard task. Implicitly, finding such a set for a SAT variant of a cryptanalysis instance is equivalent to constructing a guess-and-determine attack on the corresponding cipher [5]. Some SAT instances with a well understood structure may allow to choose a good S based on some key features of the original problem (see, e.g. [54, 62]). However, it is not always a possible or practical course of actions. In [53], it was proposed to view the process of finding good subsets for decomposing hard SAT instances as a process of minimizing a special black-box objective function. The structure and the features of this function are discussed in the following section.

3. Objective function

The objective function and its variants discussed in the present section serve the following purpose: for a specific SAT instance and its decomposition induced by some set of variables, to estimate the time required to solve all subproblems from the decomposition using a specific solver. Below we formalize this principle.

Let us recall that we consider a Boolean formula in CNF as a SAT instance. We denote it by C, and the Boolean variables appearing in C as X, $|X| = n$. Assume that we have a SAT solver A and an integer N that is used to denote a random sample size. The set S, $S \subseteq X$ is used to construct a decomposition $D_S[C]$ (see the previous section). We use the Monte Carlo method [47] in the following way. First, a random sample R is constructed by randomly choosing N instances from $D_S[C]$: $R = \{C[S/\beta_1], \ldots, C[S/\beta_N]\}$, $\beta_i \in \{0,1\}^{|S|}$, $i = 1, \ldots, N$. To calculate the function's value for S, the solver A is launched on each subproblem from R. The runtime of A on $C[S/\beta_i]$ (in seconds) is denoted by $T_A(C[S/\beta_i])$. Then the value of F is computed as follows:

$$F_{N,C,A}(S) = 2^{|S|} \times \frac{1}{N} \times \sum_{i=1}^{N} T_A(C[S/\beta_i]). \quad (1)$$

The value of F for the given C and S is an estimation of the time (in seconds) required to solve SAT for C via solving all subproblems from $D_S[C]$ by the solver A. Note, that any possible input S can be viewed as a Boolean vector of size n, such that its $i-th$ element is 1 if $i \in S$, and 0 otherwise. It is clear that an arbitrary Boolean vector of size

n corresponds to some S. Therefore, the function (1) maps $\{0,1\}^n$ onto $\mathbb{R}$, so (1) is a pseudo-Boolean function. The function (1) is stochastic since its values are computed using the Monte Carlo method. Since SAT solvers are essentially heuristic engines, and the fact that computing a value of (1) consists in the observation of their behaviour, it follows that there is no analytic form for (1). Also, it is very costly to compute its values because processing SAT instances from a random sample can actually take an arbitrarily large amount of time. To summarize, (1) is a *stochastic costly pseudo-Boolean black-box function*. Thus, the spectrum of optimization algorithms that can be used to optimize it is actually quite limited. We will touch this question in more detail in the next section.

Note, that a direct implementation of the outlined algorithm for computing function (1) (as it was done in, e.g. [53]) is not practical from the point of view of contemporary SAT solving techniques. The vast majority of state-of-the-art SAT solving algorithms are based on the Conflict-Driven Clause Learning (CDCL) algorithm [57] that implements depth-first search in a decision tree, augmented with backjumping, heuristics and the so-called *clause learning*. According to clause learning, when a variable value at the currently observed node of the search tree leads to derivation of information that contradicts the previous search history, this fact is used to construct a *conflict clause* which is added to a special database. The similar approach is employed by branch-and-bound algorithms [41]. Due to the fact that the subproblems in $D_S[C]$ differ only in the values of variables from S, it is possible to naturally organize the solving of groups of these instances in such a way that the information (conflict clauses) derived when solving earlier instances can be used to help solve the currently processed ones. This approach is usually referred to as *incremental* SAT solving [19] and is widely employed in Divide-and-Conquer solving of hard SAT instances (see, e.g. [31]). If we draw a parallel with Branch and Bound, it means that it is possible to reuse the information about some of the evaluated branches for one subproblem when solving another.

In practice, it means that when the instances from a decomposition are processed, they should be grouped in such a way that the profit from re-used information is maximized, i.e. instantiations of variables from S used to form them should be close to each other in $\{0,1\}^{|S|}$ to preserve *locality*. Note, that in [53] the values of (1) were computed without using incremental SAT solving, but the processing of decompositions aimed at solving the corresponding hard SAT instances were performed via incremental SAT. Thus, there was a clear diasgreement between the method used during computation of (1) and the method used during solving the subproblems from a decomposition found as a result of minimization of (1).

To make things better, it is necessary to construct random samples in a more sophisticated manner during the minimization of (1), e.g. in a way proposed in [62]. Instead of randomly choosing $\beta_i \in \{0,1\}^{|S|}$, $i = 1,\ldots,N$, we can choose $u, v << N$, such that $u \times v = N$. Then we pick $\beta_1,\ldots,\beta_u$ at random and for each β_j, $j = 1,\ldots,u$, $\beta_j \in \{0,1\}^{|S|}$, construct sets $B_j = \{\beta_j, \beta_j^1, \ldots, \beta_j^{v-1}\}$ by choosing $v-1$ points from their neighbourhoods in $\{0,1\}^{|S|}$. It is possible to define the neighbourhood by different means. The way used by us is described further. When the SAT solver A solves the subproblems, it processes them incrementally in groups centered around their respective β_j. Let us denote the time required by A to incrementally process the neighbourhood B_j by $T_A^I(B_j) = T_A^I(\{C[S/\beta_j^h], \beta_j^h \in B_j\})$. Then the modified objective function will

look as follows:

$$G_{N,C,A,u}(S) = 2^{|S|} \times \frac{1}{N} \times \sum_{j=1}^{u} T_A^I(\{C[S/\beta_j^h], \beta_j^h \in B_j\}). \quad (2)$$

Note that the Monte-Carlo method is used to calculate (2), as well as that for (1). Moreover, the function (2) can be viewed as a special case of (1). In opposite to (1), here the random sample size is u. During the minimization of (2) the subproblems from a sample are solved in the same way (incrementally) as in the process of their solving during the processing of a whole decomposition.

The method for constructing random samples in a way that makes it possible to harness the potential of incremental SAT solving was described in [39, 62]. It implies that the points β_j^h, $h = 1, \ldots, v-1$ are picked according to the following procedure. Note that if we fix the order of variables (e.g. in an ascending order), then there is a clear one-to-one correspondence between each point $\beta \in \{0,1\}^{|S|}$ and a number $0 \leq p_\beta \leq 2^{|S|}$ produced as a result of interpreting the sequence of 0 and 1 forming β as a binary number. The procedure starts from p_{β_j} and increments this number until specific boundary conditions are satisfied. So it processes $p_{\beta_j}, p_{\beta_j}+1, p_{\beta_j}+2, \ldots$, etc. An important fact is that it is possible to use a polynomial algorithm to check whether a point, corresponding to $p_{\beta_j}+k$ results in a very simple subproblem. For this purpose the SAT-solver is launched in a special mode, that applies only the Unit propagation rule [17] to such a point (in particular, we add to the original CNF C the unit clauses, corresponding to values defined by point β and apply Unit propagation rule to the obtained SAT instance). Since the processing is performed incrementally, it essentially works as a DPLL algorithm [17]. As a result, the procedure constructs a sample which has only nontrivial subproblems in it, thus greatly improving the quality of the Monte Carlo estimation. In particular, it allows to deal with the situations, when the value of a variable is implied by a partial assignment of several other variables. For example, if S contains such x_1, x_2, x_3 for which in the SAT instance C there is (among others) the subformula equivalent to $x_1 \oplus x_2 \oplus x_3 \equiv 1$ ($\oplus$ is the addition modulo 2 so the formula can be read as "sum of x_1, x_2, and x_3 modulo 2 should always be equal to 1"), then the effective size of decomposition $D_S[C]$ is $\leq 2^{|S|-1}$. Thus, if one constructs a random sample by picking assignments at random from $\{0,1\}^{|S|}$, in about half the cases there will be assignments that contain the values of x_1, x_2, x_3 such that $x_1 \oplus x_2 \oplus x_3 \equiv 0$ results in an easy contradiction. The equations of this kind can be found in many cryptographic algorithms, such as Bivium and Trivium [10]. The procedure from [39, 62] makes it possible to filter out the assignments of variables corresponding to trivial subproblems of this kind and put into the random sample only the subproblems that are not trivially resolved.

In [62], we analyzed the alternating step (ASG) generator [25] by SAT-based cryptanalysis. We employed a simple tabu search-based algorithm to minimize functions (1) and (2) and used the found sets to solve several cryptanalysis instances for ASG with 72-bit and 96-bit secret keys. The estimations obtained using (1) turned out to be highly inaccurate: sometimes they were several times larger, and sometimes lower, than the real runtime. The results obtained using (2), augmented with preprocessing of random samples, yielded the sets with better estimations and for which the difference between the real runtime and the estimation was less than 20% in all cases. Based on this we make a conclusion that the function (2) augmented with additional processing of random samples allows to significantly improve the quality of estimations provided

by objective function's values, while not requiring significant additional resources. According to these results, function (2) is more accurate than function (1). In this study, only function (2) is used in application to hard SAT-based optimization problems.

4. Optimization algorithms

The objective function G proposed in Section 3 can be minimized by any combinatorial black-box optimization algorithm. In this section, we consider 8 algorithms of different types that are suitable for this purpose:

(1) simple random search (SRS);
(2) oriented random search (ORS);
(3) simple hill climbing (SHC);
(4) steepest ascent hill climbing (SAHC);
(5) tabu search (TS);
(6) hill climbing with variables-based jump (HCVJ);
(7) (1+1) evolutionary algorithm ((1+1)-EA);
(8) sequential model-based algorithm configuration (SMAC).

Some features of these algorithms are shown in Table 1. Here we used the classification from the survey [7]. The term *memory usage* means that the algorithm uses either short or long memory to store the search history. The term *trajectory* means that a successor solution is sought in a neighbourhood of the current solution.

Table 1. Some features of the optimization algorithms chosen to minimize the objective function.

Algorithm	Stochastic	Memory usage	Trajectory-based
SRS	Yes	No	No
ORS	Yes	No	No
SHC	No	No	Yes
SAHC	No	No	Yes
TS	No	Yes	Yes
HCVJ	No	Yes	Yes
(1+1)-EA	Yes	No	No
SMAC	Yes	Yes	No

Assume that C is a CNF over a set of Boolean variables X, $|X| = n$. Hereinafter by a *point* of the search space we mean a set S, $S \subseteq X$. By G_{best} and S_{best} the best found value of the objective function G and the corresponding point are denoted. Let us give an overview of the employed algorithms below.

The *simple random search* algorithm randomly chooses points in the whole search space and calculates objective function values for them. G_{best} and S_{best} are updated if a new value is better than the current best known one. The pseudo-code is shown in Algorithm 1.

The *oriented random search* algorithm starts from the given starting point S_{start}, in the role of which the whole set X can be used. The value of G is calculated for S_{start}, then its value is calculated for randomly chosen points of size $|S_{start}| - 1$ until at any of them G_{best} is updated or the time limit is exceeded. If G_{best} is updated, then the algorithm starts processing randomly chosen points of size $|S_{start}| - 2$ and so on. As a result, the algorithm is oriented to decreasing the size of the best point.

Algorithm 1: Simple random search

Input: CNF C over X, solver A, random sample size N, number of intervals u, time limit t

Output: S_{best} with the runtime estimation G_{best}

```
1  S_best ← ∅
2  G_best ← 0
3  repeat
4      S ← RandomlyChoosePoint(X)
5      g = G_{N,C,A,u}(S)
6      if g < G_best or S_best = ∅ then
7          G_best ← g
8          S_best ← S
9  until TimeExceeded(t)
10 return ⟨S_best, G_best⟩
```

The next four algorithms are all trajectory-based: *simple hill climbing*, *steepest ascent hill climbing*, *tabu search*, and *hill climbing with variables-based jump*. They all operate with neighbourhoods of points from the search space. Each possible subset S of the set of Boolean variables $X, |X| = n$, corresponds to a Boolean vector of length n (see Section 3). It is quite natural to operate with the Hamming distance [26] between the corresponding Boolean vectors to form a neighbourhood. Further, a neighbourhood of a given point S is defined as a set of points, for which their representation as Boolean vectors are at Hamming distance at most H from the Boolean representation of S. The value of H used in our experiments is discussed in Section 6.

Algorithm 2: Simple hill climbing

Input: CNF C over X, time limit t, solver A, random sample size N, number of intervals u, starting point S_{start}, Hamming distance H

Output: S_{best} with the runtime estimation G_{best}

```
1  S_best ← S_start
2  G_best ← G_{N,C,A,u}(S_start)
3  S_centre ← S_start
4  repeat
5      NewOptimum ← false
6      for each S ∈ GetNeigbourhood(S_centre, H) do
7          g = G_{N,C,A,u}(S)
8          if g < G_best then
9              G_best ← g
10             S_best ← S
11             S_centre ← S
12             NewOptimum ← true
13             Break
14 until TimeExceeded(t) or NewOptimum = false
15 return ⟨S_best, G_best⟩
```

The *simple hill climbing* algorithm (see, e.g. [52]), processes points from the neighbourhood of a given starting point in some order. It calculates the objective function's value for each point and as soon as it finds a point with a better function value, the algorithm immediately starts checking the neighbourhood of this better point and so on. If all points from a neighbourhood are worse than the current S_{best}, then a local

minimum is reached and the algorithm stops. Algorithm 2 shows its pseudo-code. The function *GetNeigbourhood(S,H)* returns all points from the neighbourhood of S, which are at Hamming distance at most H from it.

The only difference between *simple hill climbing* and *steepest ascent hill climbing* (see, e.g. [52]) is that the latter calculates objective function values for all points from a current neighbourhood even if G_{best} has been already updated in this neighbourhood. Thus it always transitions to the best point in a neighborhood. The pseudo-code of this algorithm can be obtained by removing line 13 from Algorithm 2.

The Tabu search was proposed in [24]. We implemented a *tabu search* algorithm on the basis of the steepest ascent hill climbing. A tabu list is used, where the last 1000 best points from the checked neighbourhoods are stored. If a local minimum is reached, the search does not stop. Instead, the best point from the neighbourhood is chosen, and the the processing of its neighbourhood is started. Algorithm 3 shows the corresponding pseudo-code. The function *GetAdmittedNeighbours* returns all points from the neighbourhood of a given point, which are not in the tabu list. The function *UpdateTabuList* adds a given point to the top of the tabu list. If the limit on the tabu list size is reached, *UpdateTabuList* removes the last point from the tabu list before adding a new one. By $S_{candidate}$ and $G_{candidate}$ we denote the best admitted point from the current neighbourhood and the function's value for it.

Algorithm 3: The tabu search algorithm

Input: CNF C over X, time limit t, solver A, random sample size N, number of intervals u, starting point S_{start}, Hamming distance H

Output: S_{best} with the runtime estimation G_{best}

```
1  S_best ← S_start
2  G_best ← G_{N,C,A,u}(S_start)
3  S_centre ← S_start
4  repeat
5      AdmittedPoints ← GetAdmittedNeighbours(S_centre, H, TabuList)
6      if AdmittedPoints = ∅ then
7          Break
8      S_candidate ← ∅
9      G_candidate = G_best
10     for each S ∈ AdmittedPoints do
11         g = G_{N,C,A,u}(S)
12         if S_candidate = ∅ or g < G_candidate then
13             S_candidate ← S
14             G_candidate ← g
15     if G_candidate < G_best then
16         G_best ← G_candidate
17         S_best ← S_candidate
18     UpdateTabuList(S_candidate, TabuList)
19     S_centre ← S_candidate
20 until TimeExceeded(t)
21 return ⟨S_best, G_best⟩
```

The *hill climbing with variables-based jump* algorithm was proposed in [39, 63]. It is the simple hill climbing algorithm, improved by a jump strategy that helps to escape

local minima. According to this strategy, for each of n Boolean variables from X two counters are used. The first one counts how many times a variable occurred in points, for which the objective function was calculated. The second one counts how many times a variable occurred in points in which S_{best} was updated. If a local minimum is reached, then a new starting point is constructed by adding $2m$ Boolean variables to the current S_{best}. Among them, m variables are the ones with the lowest values of the first counter. In the role of another m variables the ones with the greatest value of the second counter are chosen. As a result, new variables are tried, which can be quite promising. On the other hand, successful variables are added. In all experiments described in Section 6, m was equal to 4. The pseudo-code is shown in Algorithm 4. Functions *UpdateFirstCounter* and *UpdateSecondCounter* update the first and second counter respectively. The function *ConstructNewStartPoint* constructs a new point as it was described above. Additionally, all points for which the objective function was calculated are added to a tabu list, because this algorithm is oriented on the function G which is extremely costly.

Algorithm 4: Hill climbing with variables-based jump

Input: CNF C over X, time limit t, solver A, random sample size N, number of intervals u, starting point S_{start}, jump paramter m, Hamming distance H
Output: S_{best} with the runtime estimation G_{best}

$S_{best} \leftarrow S_{start}$
$G_{best} \leftarrow G_{N,C,A,u}(S_{start})$
$S_{centre} \leftarrow S_{start}$
repeat
  AdmittedPoints $\leftarrow$ `GetAdmittedNeighbours`(S_{centre}, H)
  if AdmittedPoints $= \emptyset$ **then**
    `Break`
  NewOptimum $\leftarrow$ false
  for each $S \in$ AdmittedPoints **do**
    $g = G_{N,C,A,u}(S)$
    `UpdateTabuList`(S)
    `UpdateFirstCounter`(S)
    if $g < G_{best}$ **then**
      $G_{best} \leftarrow g$
      $S_{best} \leftarrow S$
      $S_{centre} \leftarrow S$
      `UpdateSecondCounter`(S_{best})
      NewOptimum $\leftarrow$ true
      `Break`
  if NewOptimum $=$ true **then**
    $S_{centre} \leftarrow S_{best}$
  else
    $S_{centre} \leftarrow$ `ConsrtuctNewStartPoint`(S_{best}, m)
until `TimeExceeded`(t)
return $\langle S_{best}, G_{best} \rangle$

An *(1+1) evolutionary* algorithm was implemented in accordance with its description from [18]. The only difference is that here it starts from a given point. The

function *BoolVecFromPoint* returns a Boolean vector representation (of size n) of a given point S. The function *PointFromBoolVec* returns a point from a given Boolean vector of size n. The function *RandomlyFlipVectorElements* flips independently each element of a given Boolean vector of size n with probability $1/n$. Algorithm 5 shows the pseudo-code.

Algorithm 5: The (1+1) evolutionary algorithm

Input: CNF C over X, solver A, random sample size N, number of intervals u, starting point S_{start}, time limit t
Output: S_{best} with the runtime estimation G_{best}

$S_{best} \leftarrow S_{start}$
$G_{best} \leftarrow G_{N,C,A,u}(S_{start})$
$S_{centre} \leftarrow S_{start}$
repeat
  $x \leftarrow$ `BoolVecFromPoint`(S_{best})
  $x' \leftarrow$ `RandomlyFlipVectorElements`(x)
  $S \leftarrow$ `PointFromBoolVec`(x')
  $g = G_{N,C,A,u}(S)$
  if $g < G_{best}$ **then**
    $G_{best} \leftarrow g$
    $S_{best} \leftarrow S$
until `TimeExceeded`(t)
return $\langle S_{best}, G_{best} \rangle$

Sequential model-based algorithm configuration (SMAC) [34] is an implementation of the sequential model-based optimization (SMBO) framework [38]. According to SMBO, a regression model is constructed that predicts values of an objective function and then this model is used for optimization. SMAC is based on the *random forest* machine learning algorithm [9]. In SMAC, random forest contains regression trees that have real values (values of the objective function) at their leaves. Every time a new value of the objective function is calculated, a random forest is reconstructed. In fact, random forest recommends the points for which the objective function should be calculated. SMAC can be used either for tuning algorithm parameters or for optimizing costly black-box functions.

5. Considered problems

To compare algorithms from Section 4 when minimizing the objective function (2), we considered four hard optimization problems. Each of them is related to a certain stream cipher. The required background on such relation as well as the descriptions of the optimization problems are presented below.

Any stream cipher (with some reservations) can be considered as a discrete function that transforms an input called *secret key* into an output called *keystream* [46]. Stream ciphers are usually used to quickly encrypt large amount of online data, e.g. phone calls. The encryption is performed by combining the produced keystream with a plain text (an original data). Stream ciphers are designed in such a way, that given a secret key, the corresponding keystream is produced very fast. Moreover, the algorithms are created specifically to make it extremely hard to find the input corresponding to

any given output (i.e. such discrete functions are hard to invert). Each stream cipher consists of one or several registers, where the internal state of the cipher is stored. First, given a secret key, the initialization phase is performed to fill the registers with nontrivial contents. Then, using the registers' state, the keystream is produced. Usually, the following variant of the cryptanalysis of stream ciphers is considered: given a known keystream fragment to find the registers' initial state that was used to produce this keystream fragment (see, e.g. [44]). Basically, it allows one to forgo the initialization phase at a cost of having to find the unknown values for the whole contents of ciphers registers (at the beginning of keystream generation) instead of the *secret key* itself (which is typically smaller than the size of registers). In the present study, this very variant of the cryptanalysis is considered in application to the following stream ciphers: Trivium [10], Grain_v1 [29], Mickey [3], and Rabbit [8]. They are the finalists of the eSTREAM project [12] that was completed in 2008. This project was organized by European cryptological community and was aimed at identifying new fast and resistant stream ciphers. Thus, the eSTREAM's finalists can be considered as state of the art in the area of stream ciphers.

The characteristics of the considered stream ciphers are presented in Table 2, together with sizes of the analyzed keystream fragments.

Table 2. Characteristics of the considered stream ciphers. All sizes are presented in bits.

Cipher	Secret key size	Registers size	Analyzed keystream size
Trivium	80	288	300
Grain_v1	80	160	200
Mickey	80	200	250
Rabbit	128	513	512

We studied the cryptanalysis of outlined ciphers in SAT form. This type of cryptanalysis is called *SAT-based cryptanalysis* [15, 43]. According to it, an original problem is reduced to SAT by generating the corresponding CNFs which are in turn processed (after some additional modifications) by some SAT solving algorithm.

It is possible to construct CNFs for SAT-based cryptanalysis directly or using the following tools: CBMC [11]; URSA [37]; Transalg [48]; CryptoSAT [40]. In the present paper, we used the Transalg tool because it recently showed good results in application to SAT-based cryptanalysis of stream ciphers [56]. Transalg uses a C-like language to describe a considered problem. Thus, thanks to the fact that eSTREAM candidates provided a C-implementation for each generator, it was relatively simple to transform the latter into programs for Transalg and test their correctness. Details on obtained CNFs are presented in Table 3.

Table 3. Characteristics of CNFs of the considered cryptanalysis problems.

Cipher	Variables	Clauses	Size (Mb)
Trivium	1 887	22 776	0.5
Grain_v1	2 425	47 702	1.4
Mickey	72 078	586 080	15.7
Rabbit	98 449	879 713	23.7

Note, that each of these CNFs encodes the corresponding algorithm of a stream cipher. The constructed CNFs, once the values of the keystream bits are fixed in them by assigning values to corresponding Boolean variables, form cryptanalysis instances. They can be viewed as very hard combinatorial problems. The function (2) together

with the algorithms we use to minimize it, makes it possible to estimate the time required to solve these problems. It means, that as result of minimizing objective function (2), it is possible to obtain valuable results (from cryptographic point of view). Thus, we studied 4 hard stochastic pseudo-Boolean black-box optimization problems.

It is important to notice that each CNF from Table 3 has more than a thousand variables. However, the fact that we consider the cryptanalysis instances makes it possible to reduce the search space quite significantly. Note that the initial values of registers' state (which we actually have to find in order to solve a cryptanalysis problem) are more important compared to all the other Boolean variables in the formula. Let us refer to them as to *input variables*. It is clear that the values of all the other variables in CNF formula depend on the values of input variables because the cryptographic algorithm basically transforms them this way and that in a multistage manner to finally produce keystream. Thus, we can greatly reduce the search space by limiting it to only include all possible subsets of a set of input variables. A side benefit of this treatment is that the hardness of produced subproblems is usually quite uniform, leading to a better consistency of estimations.

Thus, in the computational experiments the objective function (2) is minimized over the search spaces of the following sizes: 2^{288} for Trivium; 2^{160} for Grain_v1; 2^{200} for Mickey; 2^{513} for Rabbit.

Note, that the proposed objective function can be applied not only to SAT-based cryptanalysis. In [39], we successfully employed it for solving hard SAT instances used in SAT competitions to compare parallel SAT solvers. In particular, a Divide-and-Conquer approach based on the optimization of function (2) showed good effectiveness on several classes of hard crafted benchmarks.

6. Computational experiments

The proposed objective function G (see Section 3) was implemented as a module of the ALIAS tool [39]. This tool is aimed at solving hard SAT instances via the partitioning approach described in Section 2. The guide on how to use the source code of the proposed objective function in application to hard SAT instances is available online [1], see the item "HOW TO USE THE OBJECTIVE FUNCTION". To calculate the objective function, ALIAS operates with incremental SAT solvers via the IPASIR interface [4]. In all experiments described below, the IPASIR-based version of the SAT solver rokk [61] was used because it recently has shown good results on SAT-based cryptanalysis problems (see, e.g. [48, 53, 55, 62]).

As for the function's parameters, in all experiments N, u, v were equal to 1000000, 100, and 10000 respectively. The Hamming distance H was equal to 1. It means that only "add/remove" type of state transition was allowed in the trajectory-based optimization algorithms (i.e. for 4 out of 8 of them, see Section 4). Indeed, $H = 2$ would allow several additional transitions, including "replace". However, according to our preliminary experiments, $H = 2$ makes it impractical to use the *steepest ascent hill climbing* and the *Tabu search* algorithms. The reason is that in both these algorithms it is required to check all points in the current neighbourhood, and for all four considered optimization problems the search spaces are too large for this. The employed function is extremely costly, that is why such neighbourhoods are way too large.

We studied four optimization problems described in Section 5. All optimization algorithms described in Section 4, except SMAC, were implemented in C++ as a part of the ALIAS module responsible for the objective function minimization. In case of

SMAC, we used its implementation in Java described in [33].

As a starting point S_{start} for every algorithm, except SRS, we used a set of Boolean variables that encodes the initial states of the corresponding stream cipher's registers (see Table 2). For example, for the MICKEY cryptanalysis it was a set of 200 Boolean variables $x_1, x_2, ..., x_{200}$. Such a starting point is very convenient, because the objective function is computed effectively on it (according to the features of stream ciphers). As a result, a baseline runtime estimation can be easily calculated. It is important because for an arbitrary point from the search space it is not always possible to compute values of functions (1)-(2) in reasonable time.

Some of the implemented optimization algorithms are not stochastic (see Section 3). However, the objective function itself is stochastic, so each optimization algorithm was run 3 times on each optimization problem to alleviate the effect of randomness. Thus, $4 \times 8 \times 3 = 96$ computational experiments were performed in total. Each experiment was launched with the time limit of 1 day on one node of the computing cluster "Academician V.M. Matrosov" [13]. Every computational node of the cluster is equipped with 2 $\times$ 18-core Intel Xeon E5-2695 CPUs and 128 Gb of RAM. Thus, each experiment was held on 36 CPU cores. At any moment of time each implemented optimization algorithm operates with exactly one point from a search space, but the implementation of the objective function is a multi-threaded program, so all 36 CPU cores are employed to calculate its value.

Since SAT solvers are essentially heuristic algorithms, they can work for a very long time during the calculation of the objective function value in some point. That is why the time limit on the SAT solver runtime was used. If for any subproblem from a random sample (see Section 3) the SAT solver's runtime exceeded the imposed time limit, the processing of the sample was interrupted with the objective function value set to plus infinity. The time limit of 10 seconds was used in all the experiments. According to the recent studies ([62, 64]) it should be sufficient. Note, that in opposite to [55], we use the runtime limit only to calculate Monte-Carlo estimations. When the found set of variables is used to solve an original problem, the runtime is not limited.

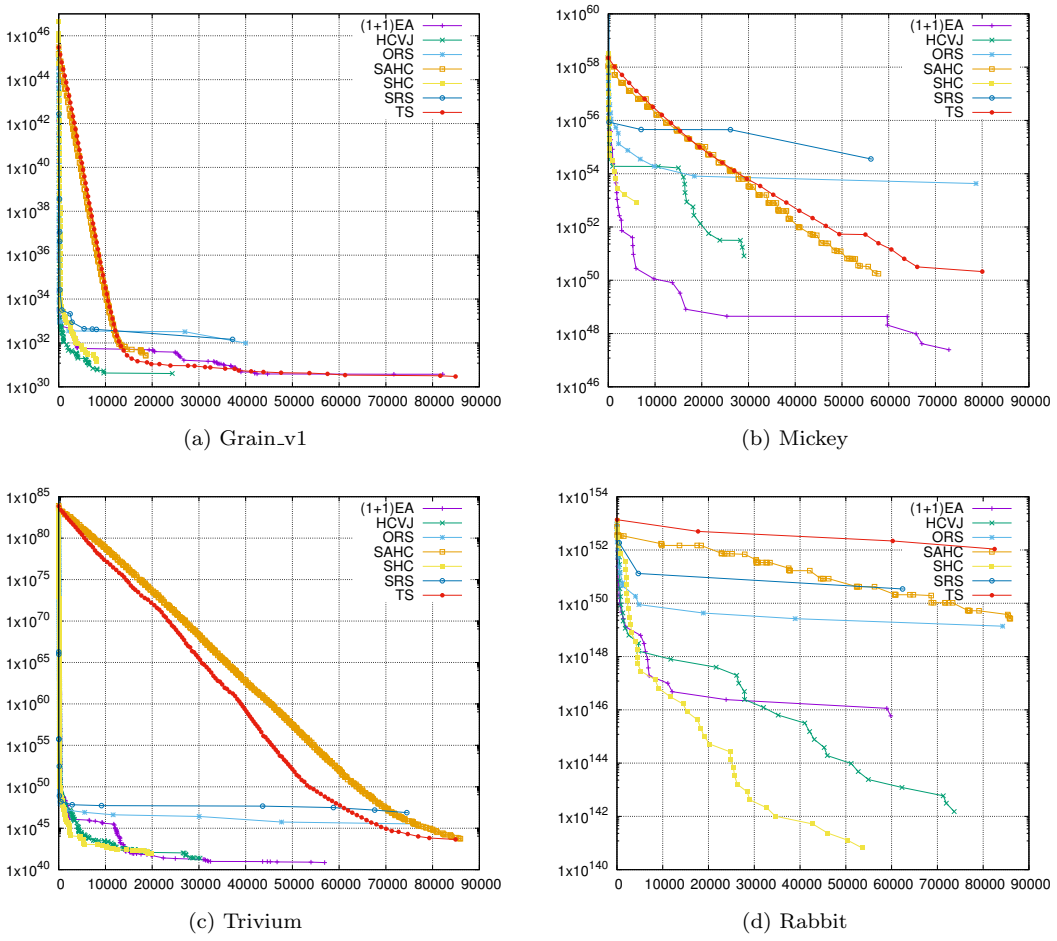
It turned out, that SMAC can not deal with the RABBIT optimization problem at all, because its maximum available objective function value is lower than baseline estimations for this cipher. SMAC uses `double` data type in Java to store the function value, so when it exceeds 1e+100, its behaviour is undefined. Of course, we could handle it by making a modified implementation that takes the logarithm of objective function's values. However, since SMAC showed quite weak results for the remaining three problems, we did not do it. That is why SMAC's results for RABBIT are absent.

For each problem, the found solution that corresponds to the best run among all optimization algorithms is presented in Appendix A. In Appendix B, the detailed results for every run are shown. In Table 4, for each pair (algorithm, problem) the result of the best run (out of 3) is shown. In Figure 1, the minimization process for the best run of each optimization algorithm (except SMAC) is shown. Here x-axis corresponds to the time in seconds elapsed since the algorithm start, while y-axis corresponds to the objective function's values for S_{best}.

Let us discuss the obtained results. Both considered random search algorithms (SRS and ORS) showed quite weak results. ORS outperforms SRS on all problems, but its results are not competitive either. However, these algorithms can be used as a baseline to distinguish the algorithms which are better than random search from those that are not. Further "algorithm A is better than algorithm B" means that the value of the objective function, found by A in its best run (out of 3) is lower than that for algorithm B. It makes sense to say that an algorithm *suits* for a certain problem, if it is better

Table 4. Results of the best runs (out of 3) for each pair (algorithm, problem). The best result for every problem among all algorithms is marked with bold.

Algorithm	Grain_v1	Mickey	Trivium	Rabbit
SRS	1.46e+32	3.58e+54	7.67e+46	3.44e+150
ORS	9.88e+31	4.29e+53	3.34e+45	1.38e+149
SHC	1.36e+31	8.2e+52	8.59e+41	**6.5e+140**
SAHC	2.58e+31	1.75e+50	5.43e+43	2.64e+149
TS	**2.96e+30**	2.11e+50	4.46e+43	1.08e+152
HCVJ	4.04e+30	8.18e+50	2.46e+41	1.52e+142
(1+1)	3.73e+30	**2.46e+47**	**7.15e+40**	5.91e+145
SMAC	1.98e+33	1.36e+54	2.12e+46	-



(a) Grain_v1 (b) Mickey

(c) Trivium (d) Rabbit

Figure 1. Minimization of the objective function on the considered problems. The x-axis corresponds to the time in seconds elapsed since the algorithm start, y-axis corresponds to the objective function's values for S_{best}.

than both SRS and ORS. It turned out, that SMAC is unsuitable for all considered problems. As for the remaining 5 algorithms (SHC, SAHC, TS, HCVJ, (1+1)-EA), their results showed that the algorithms diversity makes sense for the problems of the considered type. SAHC showed competitive results, but it does not suit for hard and relatively large problems, like Rabbit. TS found the best value of the objective function on Grain_v1, but it does not suit for Rabbit too. The main reason is that both SAHC and TS thoroughly traverse neighbourhoods (see Section 4). In the case of

Rabbit, the neighbourhoods are large, and the calculations of the objective function in the corresponding points takes a lot of time, because the original problem itself is extremely hard for SAT solvers. Note, that if we used $H = 2$, SAHC and TS would show much worse results.

It turned out, that SHC and SAHC usually do not reach the time limit on the experiment (24 hours) because both algorithms stop once a local minimum is found. Despite its simplicity, SHC showed competitive results, and it also found the best value of the objective function on Rabbit. Note, that this was achieved in a short time, because SHC stopped after finding a local minimum in all three runs and it happened before reaching the time limit. HCVJ and (1+1)-EA are the two algorithms that showed competitive results on each problem. It seems, that further improvements of SHC by various jumping strategies has great potential for the considered class of problems. Overall, the best results were shown by (1+1)-EA – it found the best values of the objective function for 2 out of 4 considered problems.

According to the best obtained estimations, all considered problems are way too hard, so the accuracy of the estimations can not be verified in reasonable time. Following [53], we studied weakened SAT-based cryptanalysis problems for the considered stream ciphers by assigning correct values to a portion of input variables. In particular, we assigned values to k (out of n) last input variables. In such a weakened variant, the first $n - k$ input variables are unknown, thus, a set S_{best} is picked from the set of all possible subsets of $\{x_1, \ldots, x_{n-k}\}$. Our goal was to find such weakened variants that, according to estimations, can be solved faster than in one day on a single cluster node. The second requirement was that $n - k \geq 64$ because we wanted to study cryptanalysis problems that are too hard for brute force. To find the estimations we used the (1+1)-EA algorithm. It was launched for 1 hour on each considered weakened variant. It turned out that for Rabbit and Mickey the suitable weakened variants correspond to $n-k < 64$, that is why we did not study them further. As for Grain_v1 and Trivium, we found suitable weakened variants with $k = 96$ and $k = 134$ respectively, i.e. S_{best} was picked from subsets of $\{x_1, \ldots, x_{64}\}$ and $\{x_1, \ldots, x_{154}\}$. The found estimation for Grain_v1 is 842477 seconds for 1 CPU core, i.e. 23402 seconds or 6 hours 30 minutes for 36 CPU cores. The corresponding set contains the following 22 variables (numeration from 1): 9, 11, 12, 13, 19, 21, 25, 26, 27, 29, 39, 43, 44, 45, 48, 52, 53, 56, 59, 60, 62, 64. Using this set, three randomly generated weakened SAT-based cryptanalysis problems were successfully solved on one cluster node. It means that satisfying assignments were found from which the correct values of the first 64 input variables were extracted. The maximum runtime was 4 hours 44 minutes with 78% of the subproblems out of 2^{22} being processed. As for Trivium, the found estimation is 368648 seconds for 1 CPU core, i.e. 10240 seconds or 2 hours 51 minutes for 1 cluster node. The corresponding set consists of the following 26 variables: 7, 10, 23, 25, 34, 38, 40, 49, 61, 62, 63, 64, 65, 67, 75, 77, 78, 103, 104, 105, 106, 120, 121, 130, 142, 151. Three weakened SAT-based cryptanalysis problems were solved on one cluster node, the maximum runtime was 1 hour 22 minutes with 51% of the subproblems being processed. Since all such estimations correspond to processing 100% of subproblems, the estimations for both analyzed weakened problems are accurate.

7. Discussion

Since a lot of optimization problems in practice do not have clearly defined objective functions, or imply that they are specified by unconventional means, the black-box

optimization methods have been flourishing in the last several decades. However, a lot of them are designed to deal only with continuous variables. For instance, it holds true for Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [27], Hooke-Jeeves method [32], pseudo-gradient approach [21] and a variety of coordinate descent techniques. Nevertheless, there are plenty of search strategies that work well with discrete variables. Generally speaking, each objective function considered in this study can be minimized by genetic algorithms [60], as well as variable neighborhood search methods [28] or other discrete black-box optimization algorithms (see, e.g. [7, 23]). In the future we plan to extend the spectrum of applicable algorithms and study how they work with optimizing functions that estimate the runtime of hard SAT instances.

The attempts to bring optimization methodology into SAT-based cryptanalysis are also nothing new. In [42], a stochastic local search algorithm was used to solve hard SAT-based cryptanalysis for the DES block cipher. Implicitly, the objective function (1) was employed to analyze the GOST block cipher and several keystream generators (Crypto-1, Hitag2, A5/1, Bivium) in [16, 20, 45, 54, 58], but in these papers the sets for decomposing an original SAT instance were constructed manually. In [53], the SAT-based cryptanalysis of the A5/1, Bivium and Grain stream ciphers were considered as optimization problems, which in turn were solved by a tabu search algorithm. In fact, the objective function minimized in that paper is the objective function (1). In [55], a completely different objective function that is constructed specifically for cryptographic problems was minimized by the optimization algorithm from [53] to analyze the Magma and AES-128 block ciphers and also the Trivium stream cipher. In [49, 50], (1+1)-EA and genetic algorithms were used to minimize the objective function from [55] in application to different weakened variants of Trivium.

The best runtime estimation for SAT-based cryptanalysis of Trivium, proposed in the present study (7.15e+40 seconds) is slightly better than the previous best such estimation, described in [55] (2.04e+41 seconds). As for Grain_v1, Mickey and Rabbit, their SAT-based cryptanalysis have not been studied before by a Divide-and-Conquer SAT approach.

8. Conclusion

In the present paper, an improvement on the pseudo-Boolean black-box optimization for SAT solving was made. In particular, we proposed a new objective function for estimating the runtime in the context of Divide-and-Conquer SAT solving, which is more accurate compared to predecessors. Several black-box optimization algorithms of different types were implemented and applied to study the proposed function on four hard optimization problem. Each of the latter is related to a problem of SAT-based cryptanalysis of a relevant stream cipher. A meticulous computational study showed that some considered optimization algorithms do not suit to the considered problems at all, while some of them show surprisingly good results.

We hope that the present paper will become a bridge between the Divide-and-Conquer SAT community from the one side, and the optimization community from the other side. The former may get access to a wide spectrum of black-box optimization strategies, while the latter can get new classes of interesting hard optimization problems, which can be used to test and compare new optimization algorithms.

The present paper is a significant extension of the papers [63] and [62] published in the proceedings of the OPTIMA'2018 and ISC'2017 conferences.

Acknowledgements

We thank anonymous reviewers for their thoughtful and constructive comments that made it possible to significantly improve the quality of the present paper. We also thank Dr. Alexander Semenov for valuable preliminary discussions.

Funding

The research was partially supported by Council for Grants of the President of the Russian Federation (grant no. MK-4155.2018.9, stipend SP-2017.2019.5) and by Russian Foundation for Basic Research (grant no. 19-07-00746-a).

References

[1] ALIAS: a modular tool for finding backdoors for SAT, https://github.com/nauchnik/alias.

[2] C. Audet and W. Hare. *Derivative-Free and Blackbox Optimization*. Springer Series in Operations Research and Financial Engineering. Springer International Publishing, Berlin, 2017.

[3] Steve Babbage and Matthew Dodd. The MICKEY Stream Ciphers. In Robshaw and Billet [51], pages 191–209.

[4] Tomás Balyo, Armin Biere, Markus Iser, and Carsten Sinz. SAT race 2015. *Artif. Intell.*, 241:45–65, 2016.

[5] Gregory V. Bard. *Algebraic Cryptanalysis*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[6] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[7] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, September 2003.

[8] Martin Boesgaard, Mette Vesterager, and Erik Zenner. The Rabbit Stream Cipher. In Robshaw and Billet [51], pages 69–83.

[9] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[10] Christophe De Cannière and Bart Preneel. Trivium. In Robshaw and Billet [51], pages 244–266.

[11] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer Berlin Heidelberg, 2004.

[12] eSTREAM: the ECRYPT stream cipher project, http://www.ecrypt.eu.org/stream/.

[13] Irkutsk supercomputer center of SB RAS, http://hpc.icc.ru.

[14] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158, 1971.

[15] Stephen A. Cook and David G. Mitchell. Finding hard instances of the satisfiability problem: A survey. In *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–18, 1996.

[16] Nicolas T. Courtois, Jerzy A. Gawinecki, and Guangyan Song. Contradiction immunity and guess-then-determine attacks on GOST. *Tatra Mountains Mathematical Publications*, 53(1):2–13, 2012.

[17] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[18] Stefan Droste, Thomas Jansen, and Ingo Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theor. Comput. Sci.*, 276(1-2):51–81, 2002.

[19] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

[20] T Eibach, E Pilz, and G Völkel. Attacking Bivium using SAT solvers. In H K Bning and X Zhao, editors, *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing: 12-15 May 2008; Guangzhou, China*, pages 63–76, 2008.

[21] Yu. G. Evtushenko, S. A. Lurie, M. A. Posypkin, and Yu. O. Solyaev. Application of optimization methods for finding equilibrium states of two-dimensional crystals. *Computational Mathematics and Mathematical Physics*, 56(12):2001–2010, 2016.

[22] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[23] Michel Gendreau and Jean-Yves Potvin. *Handbook of Metaheuristics*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

[24] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & OR*, 13(5):533–549, 1986.

[25] C. G. Günther. *Alternating Step Generators Controlled by De Bruijn Sequences*, pages 5–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988.

[26] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950.

[27] Nikolaus Hansen and Andreas Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Proceedings of 1996 IEEE International Conference on Evolutionary Computation,*, pages 312–317, 1996.

[28] Pierre Hansen and Nenad Mladenovi. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449 – 467, 2001.

[29] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. The Grain Family of Stream Ciphers. In Robshaw and Billet [51], pages 179–190.

[30] Marijn J. H. Heule, Oliver Kullmann, and Armin Biere. Cube-and-conquer for Satisfiability. In *Handbook of Parallel Constraint Reasoning*, pages 31–59. Springer, 2018.

[31] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. In *Hardware and Software: Verification and Testing*, volume 7261 of *Lecture Notes in Computer Science*, pages 50–65, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[32] Robert Hooke and T. A. Jeeves. " direct search" solution of numerical and statistical problems. *J. ACM*, 8(2):212–229, April 1961.

[33] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION-5*, page 507523, 2011.

[34] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*, volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer, 2011.

[35] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. A distribution method for solving SAT in grids. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, volume 4121 of *Lecture Notes in Computer Science*, pages 430–435, 2006.

[36] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Partitioning SAT instances for distributed solving. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2010.

[37] Predrag Janicic. URSA: a system for uniform reduction to SAT. *Logical Methods in Computer Science*, 8(3):1–39, 2012.

[38] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *J. Global Optimization*, 13(4):455–492, 1998.

[39] Stepan Kochemazov and Oleg Zaikin. ALIAS: A modular tool for finding backdoors for SAT. In *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 419–427, 2018.

[40] Frédéric Lafitte. Cryptosat: a tool for sat-based cryptanalysis. *IET Information Security*, 12(6):463–474, 2018.

[41] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Oper. Res.*, 14(4):699–719, 1966.

[42] Fabio Massacci. Using Walk-SAT and Rel-SAT for cryptographic key search. In *IJCAI'99*, pages 290–295, 1999.

[43] Fabio Massacci and Laura Marraro. Logical cryptanalysis as a SAT problem. *J. Autom. Reasoning*, 24(1/2):165–203, 2000.

[44] Alexander Maximov and Alex Biryukov. Two trivial attacks on trivium. In Carlisle Adams, Ali Miri, and Michael Wiener, editors, *Selected Areas in Cryptography*, pages 36–55, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[45] Cameron Mcdonald, Chris Charnes, and Josef Pieprzyk. Attacking Bivium with MiniSat. Technical Report 2007/040, ECRYPT Stream Cipher Project, 2007.

[46] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[47] Nicholas Metropolis and Stanislaw Ulam. The Monte Carlo Method. *J. Amer. statistical assoc.*, 44(247):335–341, 1949.

[48] Ilya Otpuschennikov, Alexander Semenov, Irina Gribanova, Oleg Zaikin, and Stepan Kochemazov. Encoding cryptographic functions to SAT using TRANSALG system. In *ECAI 2016 - 22nd European Conference on Artificial Intelligence*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 1594–1595. IOS Press, 2016.

[49] Artem Pavlenko, Maxim Buzdalov, and Vladimir Ulyantsev. Fitness comparison by statistical testing in construction of sat-based guess-and-determine cryptographic attacks. In Anne Auger and Thomas Stützle, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 312–320. ACM, 2019.

[50] Artem Pavlenko, Alexander Semenov, and Vladimir Ulyantsev. Evolutionary computation techniques for constructing sat-based attacks in algebraic cryptanalysis. In Paul Kaufmann and Pedro A. Castillo, editors, *Applications of Evolutionary Computation - 22nd International Conference, EvoApplications 2019, Held as Part of EvoStar 2019, Leipzig, Germany, April 24-26, 2019, Proceedings*, volume 11454 of *Lecture Notes in Computer Science*, pages 237–253. Springer, 2019.

[51] Matthew J. B. Robshaw and Olivier Billet, editors. *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*. Springer, 2008.

[52] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

[53] Alexander Semenov and Oleg Zaikin. Algorithm for finding partitionings of hard variants of boolean satisfiability problem with application to inversion of some cryptographic functions. *SpringerPlus*, 5(1):1–16, 2016.

[54] Alexander Semenov, Oleg Zaikin, Dmitry Bespalov, and Mikhail Posypkin. Parallel logical cryptanalysis of the generator A5/1 in BNB-grid system. In *11 International Conference on Parallel Computing Technologies - PaCT 2011*, volume 6873 of *Lecture Notes in Computer Science*, pages 473–483, 2011.

[55] Alexander Semenov, Oleg Zaikin, Ilya Otpuschennikov, Stepan Kochemazov, and Alexey Ignatiev. On cryptographic attacks using backdoors for SAT. In *The Thirty-Second AAAI Conference on Artificial Intelligence*, AAAI'2018, pages 6641–6648, 2018.

[56] Alexander A. Semenov, Ilya V. Otpuschennikov, Irina Gribanova, Oleg Zaikin, and Stepan

Kochemazov. Translation of algorithmic descriptions of discrete functions to SAT with applications to cryptanalysis problems. *CoRR*, abs/1805.07239, 2018.

[57] João P. Marques Silva and Karem A. Sakallah. GRASP: a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, 1996.

[58] M Soos, K Nohl, and C Castelluccia. Extending SAT solvers to cryptographic problems. In O Kullmann, editor, *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing: 30 June - 3 July, 2009; Swansea, UK*, pages 244–257, 2009.

[59] G. S. Tseitin. On the complexity of derivation in propositional calculus. In A. O. Slisenko, editor, *Studies in mathematics and mathematical logic, Part II*, pages 115–125. 1968.

[60] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, Jun 1994.

[61] Takeru Yasumoto and Takumi Okuwaga. Rokk 1.0.1. In Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo, editors, *SAT Competition 2014*, page 70, 2014.

[62] Oleg Zaikin and Stepan Kochemazov. An improved SAT-based guess-and-determine attack on the alternating step generator. In *ISC 2017*, volume 10599 of *LNCS*, pages 21–38, 2017.

[63] Oleg Zaikin and Stepan Kochemazov. Pseudo-boolean black-box optimization methods in the context of divide-and-conquer approach to solving hard SAT instances. In *OPTIMA 2018 (Supplementary Volume)*, pages 76–87. DEStech Publications, Inc., 2018.

[64] Oleg Zaikin and Stepan Kochemazov. Black-box optimization in an extended search space for SAT solving. In Michael Khachay, Yury Kochetov, and Panos M. Pardalos, editors, *Mathematical Optimization Theory and Operations Research - 18th International Conference, MOTOR 2019, Ekaterinburg, Russia, July 8-12, 2019, Proceedings*, volume 11548 of *Lecture Notes in Computer Science*, pages 402–417. Springer, 2019.

[65] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21(4):543–560, 1996.

Appendix A. Best found points for the considered problems

Hereinafter the numeration from 1 for variables is used. For Grain_v1, the best value of the objective function was found by the Tabu search algorithm. The corresponding set consists of the following 108 variables (out of 160): 2, 3, 5, 6, 7, 9, 10, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 39, 40, 41, 42, 43, 44, 45, 48, 49, 50, 51, 52, 53, 54, 55, 56, 58, 59, 60, 61, 63, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 78, 79, 82, 83, 85, 86, 87, 89, 91, 92, 93, 94, 96, 97, 103, 104, 106, 107, 109, 110, 111, 112, 113, 120, 121, 125, 127, 128, 131, 132, 134, 135, 141, 144, 145, 148, 150, 151, 152, 153, 155, 160.

For Mickey, the best value of the objective function was found by the (1+1) evolutionary algorithm. The corresponding set consists of the following 163 variables (out of 200): 1, 2, 4, 5, 7, 8, 12, 14, 16, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 35, 37, 38, 39, 41, 43, 46, 47, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 70, 71, 73, 75, 77, 78, 79, 80, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 101, 104, 105, 106, 107, 109, 110, 111, 112, 113, 114, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 151, 152, 153, 156, 158, 159, 160, 161, 162, 163, 164, 165, 167, 168, 169, 170, 171, 172, 176, 177, 179, 180, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198,

199, 200.

For Trivium, the best value of the objective function was found by the (1+1) evolutionary algorithm. The corresponding set consists of the following 142 variables (out of 288): 3, 6, 8, 9, 11, 12, 15, 17, 18, 20, 21, 23, 24, 26, 27, 30, 32, 33, 38, 39, 41, 42, 44, 45, 46, 47, 48, 50, 51, 54, 57, 59, 61, 62, 64, 65, 71, 72, 74, 75, 77, 78, 81, 84, 85, 87, 88, 89, 92, 93, 95, 101, 105, 107, 108, 110, 111, 113, 114, 117, 119, 122, 123, 125, 126, 131, 132, 134, 137, 138, 140, 146, 147, 148, 152, 153, 157, 160, 164, 165, 167, 168, 171, 173, 174, 175, 177, 182, 183, 186, 189, 192, 195, 198, 200, 201, 203, 206, 207, 210, 212, 213, 215, 216, 218, 219, 221, 222, 225, 227, 228, 230, 233, 238, 239, 240, 242, 246, 249, 251, 252, 254, 256, 257, 258, 260, 261, 262, 265, 266, 267, 270, 272, 273, 276, 281, 282, 283, 284, 285, 287, 288.

For Rabbit, the best value of the objective function was found by the simple hill climbing algorithm. The corresponding set consists of the following 472 variables (out of 513): 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 92, 93, 94, 95, 96, 97, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 110, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 124, 126, 127, 128, 131, 132, 133, 136, 140, 141, 142, 143, 144, 145, 148, 149, 152, 154, 155, 156, 157, 158, 159, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 256, 257, 258, 259, 260, 261, 262, 263, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 361, 364, 365, 368, 369, 370, 371, 372, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 479, 480, 481, 482, 483, 484, 485, 486, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513.

Appendix B. Additional figures and tables

In tables B1, B2, B3, and B4, the results of each conducted run for Grain_v1, Mickey, Trivium, and Rabbit are shown. Here "func." stands for the best found value of the objective function. These results are also shown in Figure B1.

Table B1. Objective functions values found by optimization algorithms for Grain_v1. The best result for every algorithm is marked with bold.

Alg.	Run 1		Run 2		Run 3	
	func.	time	func.	time	func.	time
SRS	**1.46e+32**	24 h	1.76e+32	24 h	3.63e+32	24 h
ORS	1.43e+32	24 h	**9.88e+31**	24 h	2.75e+32	24 h
SHC	**1.36e+31**	2 h 16 m	2.92e+31	2 h 24 m	7.97e+32	49 m
SAHC	**2.58e+31**	5 h 10 m	4.43e+31	4 h 12 m	3e+31	4 h 10 m
TS	5.23e+30	24 h	**2.96e+30**	24 h	7.38e+30	24 h
HCVJ	**4.07e+30**	24 h	4.77e+30	24 h	5.85e+30	24 h
(1+1)	**3.73e+30**	24 h	5.56e+30	24 h	7.8e+30	24 h
SMAC	1.78e+34	24 h	**1.98e+33**	24 h	1.95e+34	24 h

Table B2. Objective functions values found by optimization algorithms for Mickey. The best result for every algorithm is marked with bold.

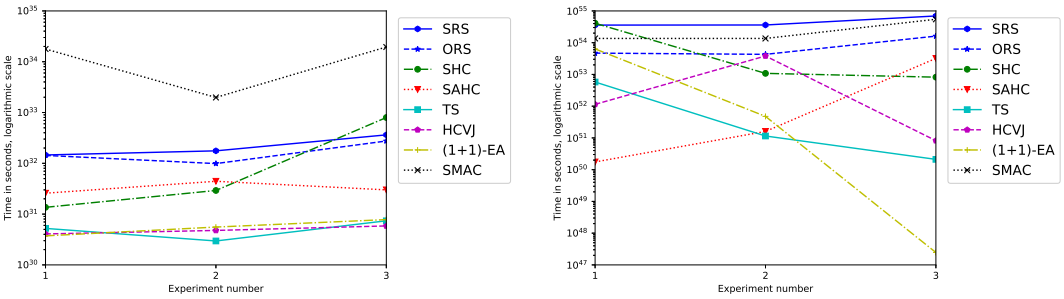
Alg.	Run 1		Run 2		Run 3	
	func.	time	func.	time	func.	time
SRS	**3.58e+54**	24 h	3.61e+54	24 h	6.94e+54	24 h
ORS	4.69e+53	24 h	**4.29e+53**	24 h	1.61e+54	24 h
SHC	4.09e+54	1 h 9 m	1.08e+53	2 h 8 m	**8.2e+52**	2 h 43 m
SAHC	**1.75e+50**	17 h 9 m	1.6e+51	19 h 51 m	3.2e+53	10 h 33 m
TS	5.77e+52	24 h	1.15e+51	24 h	**2.11e+50**	24 h
HCVJ	1.14e+52	24 h	3.81e+53	24 h	**8.18e+50**	24 h
(1+1)	6.37e+53	24 h	4.77e+51	24 h	**2.46e+47**	24 h
SMAC	**1.36e+54**	24 h	1.36e+54	24 h	5.46e+54	24 h

Table B3. Objective functions values found by optimization algorithms for Trivium. The best result for every algorithm is marked with bold.

Alg.	Run 1		Run 2		Run 3	
	func.	time	func.	time	func.	time
SRS	4.69e+47	24 h	1.7e+47	24 h	**7.67e+46**	24 h
ORS	2.39e+46	24 h	**3.34e+45**	24 h	2.29e+46	24 h
SHC	**8.59e+41**	6 h 18 m	4.91e+44	3 h 38 m	1.79e+48	1 h 40 m
SAHC	1.88e+45	22 h 35 m	5.46e+45	24 h	**5.43e+43**	24 h
TS	**4.46e+43**	24 h	1.48e+46	24 h	2.59e+45	24 h
HCVJ	5.04e+41	24 h	**2.46e+41**	24 h	1.07e+42	24 h
(1+1)	**7.15e+40**	24 h	9.11e+42	24 h	3.07e+41	24 h
SMAC	2.78e+47	24 h	4.87e+46	24 h	**2.12e+46**	24 h

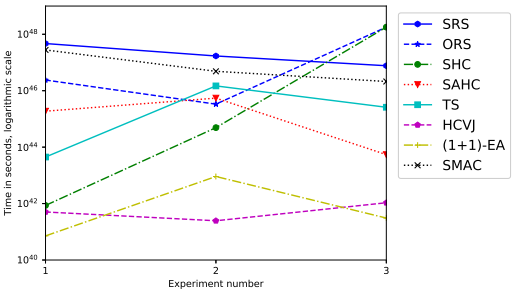
Table B4. Objective functions values found by optimization algorithms for Rabbit. The best result for every algorithm is marked with bold.

Alg.	Run 1		Run 2		Run 3	
	func.	time	func.	time	func.	time
SRS	**3.44e+150**	24 h	6.32e+150	24 h	4.25e+150	24 h
ORS	2.61e+149	24 h	**1.38e+149**	24 h	1.61e+149	24 h
SHC	1.25e+141	15h 26m	6.01e+147	6h 21m	**6.5e+140**	18h 32m
SAHC	1.15e+152	24 h	**2.64e+149**	24 h	1.27e+150	24 h
TS	1.16e+152	24 h	**1.08e+152**	24 h	1.18e+152	24 h
HCVJ	4.8e+146	24 h	**1.52e+142**	24 h	1.19e+146	24 h
(1+1)	1.25e+147	24 h	5.36e+147	24 h	**5.91e+145**	24 h
SMAC	-	-	-	-	-	-

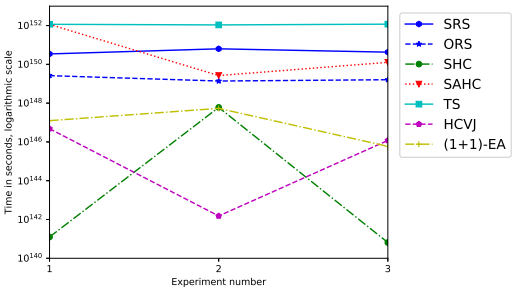


(a) Grain_v1

(b) Mickey



(c) Trivium



(d) Rabbit

Figure B1. Results of all runs of the considered optimization algorithms