Finding Effective SAT Partitionings Via Black-box Optimization

Alexander Semenov, Oleg Zaikin, and Stepan Kochemazov

Abstract In the present chapter we study one method for partitioning hard instances of the Boolean satisfiability problem (SAT). It uses a subset of a set of variables of an original formula to partition it into a family of subproblems that are significantly easier to solve individually. While it is usually very hard to estimate the time required to solve a hard SAT instance without actually solving it, the partitionings of the presented kind make it possible to naturally construct such estimations via the well-known Monte Carlo method. We show that the problem of finding a SAT partitioning with minimal estimation of time required to solve all subproblems can be formulated as the problem of minimizing a special pseudo-Boolean black-box function. The experimental part of the paper clearly shows, that in the context of the proposed approach relatively simple black-box optimization algorithms show good results in application to minimization of the functions of the described kind even when faced with hard SAT instances that encode problems of finding preimages of cryptographic functions.

1 Introduction

In modern world, many different concepts are used to tackle hard combinatorial problems. The rapid development of computational hardware in the last few decades puts a special emphasis on the methods that are able to make use of massive amount of computational processes provided by today's computers and supercomputers. One of the most straightforward approaches of this kind consists in partitioning a hard problem into a (possibly very large) family of subproblems which are much easier to solve. However, the question remains, how to construct such partitionings, and how to distinguish which of the many partitionings is better than the others.

Matrosov Institute for System Dynamics and Control Theory SB RAS, Irkutsk, Russia, e-mail: biclop.rambler@yandex.ru, zaikin.icc@gmail.com veinamond@gmail.com

In the present chapter we focus on just such questions that arise when considering hard instances of the Boolean satisfiability problem (SAT) [6]. The goal of SAT is for an arbitrary Boolean formula to answer the question whether there exists an assignment of its variables that satisfies this formula. Despite the fact that SAT is NP-complete [25], the progress in practical SAT solving in the recent few decades is nothing short of spectacular. Today, SAT solvers are routinely used to deal with many problems arising in a plethora of different areas, such as hardware verification, model checking, bioinformatics, and cryptanalysis. The major disadvantage of state-of-the-art SAT solvers is that it is impossible to predict how long will it take a solver to tackle any particular SAT instance given to it as an input, because in the worst-case scenario its runtime will be exponential in the number of variables of the input formula. And while the SAT solvers often work exceptionally well even with SAT instances over hundreds of thousands of variables, there remain hard SAT instances that are seemingly impossible to solve at the current level of technology.

In the present chapter we consider the so-called Divide-and-conquer approach to solving hard SAT instances, see, e.g., [76]. It consists in partitioning an instance into a family of subproblems and tackling these subproblems individually, with the possibility to solve them independently in parallel. There exist many ways to partition a SAT instances, see [34]. In the terminology of [34], we focus on the *plain partitioning* variant that uses a subset of variables of a SAT instance to split it into a family of simpler subproblems. We formally show that it is possible to use the Monte Carlo method [51] to estimate the time required by any deterministic complete SAT solving algorithm to solve all subproblems from such a family. This fact allows one to formulate the problem of finding a set of variables that yields a SAT partitioning with the smallest runtime estimation as the problem of optimizing a black-box function that takes as an input a set of variables of a SAT instance and outputs the corresponding runtime estimation. We consider three different objective functions of this type that differ in the way the problems from the SAT partitioning are tackled. Each of them is a pseudo-Boolean black-box costly stochastic function. Therefore, the range of suitable optimization algorithms is very limited. In particular, in the context of the chapter we consider several black-box optimization algorithms that rely only on direct calculations of an objective function (see, e.g., [41]).

In the computational experiments for several SAT-based cryptanalysis instances we aim to construct good runtime estimations and find the corresponding effective SAT partitionings. The results of experiments show that having a portfolio of optimization algorithms often helps since there is no algorithm that works better than the others on all possible inputs. We also checked that for the SAT instances for which the constructed runtime estimation is not too large, the time required to solve the corresponding SAT instances via the found partitionings agrees well with the constructed runtime estimations.

The chapter is organized as follows. In the next section, we briefly provide the basic notation regarding SAT and SAT-based cryptanalysis. Sect. 3 considers the technique employed to construct SAT partitionings, i.e. how an original problem is split into a family of subproblems. Then it proceeds with describing two main approaches to estimating the time required to solve the corresponding subproblems.

In particular, we provide strict formal justifications showing that the Monte Carlo method in its original formulation can be applied to the problems at hand. The main contribution of Sect. 4 is formed by defining three pseudo-Boolean black-box functions that evaluate the effectiveness of SAT partitionings of the considered kind. Also, it describes several heuristic improvements that make use of the peculiarities of hard SAT instances, which encode cryptanalysis problems, and the state-of-the-art SAT solving techniques. Sect. 5 describes the optimization algorithms that are further used to minimize the objective functions. Finally, in Sect. 6 we consider several hard optimization problems that consist in finding good runtime estimations for relevant cryptanalysis problems, present the results of the corresponding computational experiments and discuss them.

2 Preliminaries

Binary words are the words over the alphabet $\{0,1\}$. Let us denote by $\{0,1\}^k$ the set of all possible binary words of length k, $k \in \mathbb{N}^+$. The variables that take values from the set $\{0,1\}$ are called *Boolean*, thus, in some sources, the elements from $\{0,1\}^k$ are often called *Boolean vectors* of length k. The set of all possible binary words of an arbitrary finite length is denoted as

$$\{0,1\}^+ = \bigcup_{k=1}^{\infty} \{0,1\}^k$$

By *Boolean formula* over a set of variables $X = \{x_1, \ldots, x_k\}$ we mean an expression that is constructed using specific rules over a finite alphabet which includes the variables from X, braces and special auxiliary symbols called Boolean connectives. Usually, it is implied that the Boolean connectives form a *complete basis* [70]. Hereinafter, we consider Boolean formulas over complete bases $\{\wedge, \vee, \neg\}$ or $\{\wedge, \neg\}$, where $\wedge$ is conjunction, $\vee$ is disjunction, and $\neg$ is negation. The formulas of the kind x and $\neg x$, $x \in X$ are called *literals* (over X). A pair of literals $(x, \neg x)$ is called a *complementary pair*. A *clause* is an arbitrary disjunction of different literals among which no pair is complementary. A *Conjunctive Normal Form* (CNF) is an arbitrary conjunction of different clauses.

2.1 Boolean Satisfiability Problem (SAT)

Let F be an arbitrary Boolean formula over $X = \{x_1, \ldots, x_k\}$. One can naturally associate with F a Boolean function $f_F : \{0,1\}^k \to \{0,1\}$. An arbitrary Boolean vector of length k can then be viewed as an assignment of variables from X. Formula F is called *satisfiable* if there exists such $\alpha \in \{0,1\}^k$ that $f_F(\alpha) = 1$. Such an α is referred to as a *satisfying assignment* of F. If there are no assignments of variables

that satisfy F then the formula is called *unsatisfiable*. The *Boolean satisfiability problem (SAT)* is to determine the satisfiability of an arbitrary formula F [24]. Using the transformations described in [69], SAT for an arbitrary Boolean formula can be reduced to SAT for a formula in CNF in polynomial time. Thus, without the loss of generality, it is possible to consider SAT only in application to CNFs.

SAT is the historically first NP-complete problem. It is clear that the following problem is NP-hard [25]: for an arbitrary CNF C to find an assignment satisfying C or to prove that C is unsatisfiable. This problem is also denoted as SAT. Despite the NP-hardness, there are many subclasses and special cases of SAT for which it is possible to solve the corresponding instances in reasonable time. These facts led to the intensive development of computational algorithms for solving SAT. The resulting algorithms have been successfully applied to various problems [6].

In the present study we use only the SAT solving algorithms that are based on the *Conflict-Driven Clause Learning (CDCL)* concept [47, 46]. Today, they perform best overall in application to wide spectrum of problems from different areas. Informally, a CDCL algorithm traverses a binary tree representing the process of finding satisfying assignments of an input Boolean formula in CNF. During this process, it encounters the so-called *conflicts*, meaning that some branches of the search tree resulted in contradictions. CDCL solvers store the information about refuted branches in form of the *learnt clauses* [46]. As it follows from their name, they use the information about conflicts to direct the further traversal of the search tree. The distinctive feature of CDCL is that the algorithm is complete, therefore not only can it find an assignment of variables that satisfies the input formula, but it also can prove that such an assignment does not exist. An important fact for the further constructions consists in that due to its completeness, the runtime of CDCL is finite on any input SAT instance.

The practical implementations of CDCL usually employ many different heuristic techniques that allow them to cope with large industrial problems. Historically, one of the most successful solvers is the MINISAT solver [20] first presented in 2003. It introduced a simple framework that is easy to improve and experiment with. Even almost twenty years later, the improved versions of MINISAT are still considered to be among the best performing solvers.

When it comes to hard SAT instances where standard sequential SAT solvers might not be enough, there exist two main approaches to solve SAT in parallel. The first one is called the *portfolio* approach [29] that consists in launching different SAT solving algorithms (or the same algorithm with different parameters) on the same SAT instance in parallel. The motivation here is that since CDCL is a complete algorithm then at some point at least one of the algorithms will manage to solve the problem. However, in all but the most simple cases it is impossible to give any predictions when the problem will be solved. The second approach is called the *partitioning* approach and it implements the Divide-and-conquer strategy [76]. It consists in splitting the original problem into several (possibly very many) disjoint subproblems and solving them in parallel. In that case intuitively each subproblem should be easier to solve than the original one.

The existing parallel CDCL solvers such as Plingeling, Treengeling, Painless, Cryptominisat, and others usually implement a mix of portfolio and partitioning approaches [3]. In addition they use some specific parallel heuristics. However, they achieved only a moderate success since the speedup from the parallelization is usually far from linear. In addition to that, the parallel solvers are often not deterministic meaning that several launches of the same solver on the same problem may yield the answer in drastically different times.

2.2 SAT-based Cryptanalysis

To stimulate the progress, there is always a need for difficult benchmarks that act as challenges for new generations of SAT solving algorithms. One of the most fruitful areas that produces a lot of such benchmarks is combinatorics, including various graph related problems, algebraic problems, etc. Another prominent area that allows one to construct exceptionally difficult benchmarks is cryptanalysis [15]. In the present chapter we consider hard SAT instances that are related to the so-called *SAT-based cryptanalysis*, which is a subarea of the algebraic cryptanalysis [4].

Let us briefly consider the procedures used to transform cryptanalysis instances into SAT instances. One can view a large part of such problems in the context of a general problem to which we will refer as to *problem of inversion of a discrete function* [61].

Consider a total function:

$$f : \{0,1\}^+ \to \{0,1\}^+, \tag{1}$$

defined by some algorithm $A(f)$. Such an algorithm naturally defines a family of functions of the kind

$$f_n : \{0,1\}^n \to \{0,1\}^+, n \in \mathbb{N}^+.$$

Hereinafter, assume that for a particular n the result of $A(f)$ on an arbitrary word $\alpha \in \{0,1\}^n$ is a binary word of length m, meaning the functions of the following kind:

$$f_n : \{0,1\}^n \to \{0,1\}^m. \tag{2}$$

We will refer to functions (1) and (2) as to *discrete functions*. Let us additionally assume that the complexity of the algorithm $A(f)$ is bounded by a polynomial in n. Then the problem of inversion of a function f is formulated as follows: for known $A(f)$, n and an arbitrary $\gamma \in Range\ f_n \subseteq \{0,1\}^m$ to find such $\alpha \in \{0,1\}^n$ that $f_n(\alpha) = \gamma$.

If we view $A(f)$ as a program for a Turing machine that works with binary data [25], then it is possible to show that there exists a procedure with a complexity bounded by a polynomial in n, that given a program $A(f)$ and an arbitrary n as an input, outputs a circuit $S(f_n)$ over a basis $\{\neg, \wedge\}$ that defines function f_n. This fact is a reformulation of the Cook-Levin theorem [14, 44] in the context of a problem

of inversion of a function of the kind (1). By applying to circuit $S(f_n)$ the Tseitin transformations [69] it is possible to construct a CNF which we denote as $C(f_n)$. Let us refer to $C(f_n)$ as to *template CNF* for f_n [61].

CNF $C(f_n)$ has an important property that we will frequently use below. This property is based on the well-known *Unit Propagation rule* [19, 46]. Essentially, it is a variant of the resolution method [58], when one of the two clauses used to construct a resolvent consists of a single literal. It works as follows. Let C be an arbitrary CNF over X and l be some literal over X. Consider CNF $C' = l \wedge C$. First, remove from C' all clauses that contain literal l except the unit clause l. Then from each clause in C' containing literal $\neg l$ remove this literal. The resulting CNF C'' is equivalent to C'. The described transformation represents one iteration of Unit Propagation.

Assume that C is some CNF over X. For an arbitrary $x \in X$ and $\delta \in \{0, 1\}$ let us define the result of the substitution of $x = \delta$ in CNF C as a CNF $C|_{x=\delta}$ constructed by replacing all occurrences of x to δ and performing all possible elementary transformations [12]. Let $l_\delta(x)$, $\delta \in \{0, 1\}$ be literal $\neg x$ when $\delta = 0$ and x when $\delta = 1$. It is easy to see that CNF $l_\delta(x) \wedge C$ is satisfiable if and only if CNF $C|_{x=\delta}$ is satisfiable. Thus, the application of Unit Propagation to $l_\delta(x) \wedge C$ can be interpreted as substitution of $x = \delta$ to C. During Unit Propagation there can appear new unit clauses of the kind $l_{\delta'}(x')$. Taking into account all of the above, we say that in this case the value δ' of variable x' *is derived* from a corresponding CNF via Unit Propagation. The property of the template CNFs we mentioned earlier consists in the following.

Let f_n be a discrete function of the kind (2). Assume that $S(f_n)$ is a Boolean circuit that specifies f_n, $C(f_n)$ is the corresponding template CNF, and X is a set of Boolean variables from $C(f_n)$. Let us outline in X the sets $X^{in} = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$ formed by the variables associated with the inputs and outputs of circuit $S(f_n)$, respectively.

Lemma 1 *Suppose that $\alpha = (\alpha_1, \ldots, \alpha_n)$ is an arbitrary assignment of variables from X^{in}. Consider a CNF*

$$l_{\alpha_1}(x_1) \wedge \ldots \wedge l_{\alpha_n}(x_n) \wedge C(f_n). \tag{3}$$

The iterative application of Unit Propagation to (3) *will result in the derivation of all variables from X. In particular, it means that the values $y_1 = \gamma_1, \ldots, y_m = \gamma_m$ will be derived such that $f_n(\alpha) = \gamma$, $\gamma = (\gamma_1, \ldots, \gamma_m)$.*

The statements which are close to Lemma 1 can be found in many papers such as [38, 5, 60, 37].

It was shown in [61] that if $\gamma = (\gamma_1, \ldots, \gamma_m) : \gamma \in Range\ f_n$, then CNF

$$C(f_n, \gamma) = l_{\gamma_1}(y_1) \wedge \ldots \wedge l_{\gamma_m}(y_m) \wedge C(f_n) \tag{4}$$

is satisfiable and from its satisfying assignment one can extract the assignment α of variables from X^{in} such that $f_n(\alpha) = \gamma$.

The transition from the inversion problem for a function of the kind (2) to SAT for some CNF (4) is an essential first step of any attempt at solving cryptanalysis

problems with SAT solvers. In practice it is possible to use various software tools to perform this step: CBMC [13]; URSA [36]; SAW [11]; CryptoSAT [43]; Grain of Salt [67]. In the present chapter we use SAT encodings constructed via the Transalg software tool [53], which takes into account many features that are specific to cryptographic functions. The detailed comparison of the tools, together with a more detailed description of the SAT-based cryptanalysis method can be found in [61].

3 Decomposition Sets and Backdoors in SAT with Application to Inversion of Discrete Functions

In the present section we describe the method that we use to partition a hard SAT instance into a family of simpler subproblems. Using the notation from [34], we employ the so-called *partitioning approach*, which can be viewed as a special case of data parallelism.

Definition 1 ([35, 34])

Let C be an arbitrary CNF over a set X of Boolean variables. A *plain partitioning* of C is a set of formulas

$$G_j \wedge C, j \in \{1, \ldots, r\}$$

such that for any $i, j : i \neq j$ formula $G_i \wedge G_j \wedge C$ is unsatisfiable and

$$C \equiv G_1 \wedge C \vee \ldots \vee G_r \wedge C.$$

where $\equiv$ stands for logical equivalence.

Obviously, when one has a plain partitioning of an original SAT instance, SAT for formulas $G_j \wedge C$, $j \in \{1, \ldots, r\}$ can be solved independently in parallel. There exist various partitioning techniques. For example one can construct $\{G_j\}$, $j = 1, \ldots, r$ using the so-called scattering procedure, a guiding path solver, look-ahead solver, or a number of other techniques described in [34].

The idea to use look-ahead strategy to construct SAT partitionings, first expressed in [35], was later developed in [32], which presented a SAT solver that combines the features of CDCL and look-ahead concepts. In more detail, in [32] it was proposed to use a look-ahead solver as an external procedure that constructs some partitioning tree. If during this process the look-ahead solver refutes some branch, then this branch is not considered later. Otherwise, the look-ahead solver uses the special cutoff heuristics to terminate the construction of a tree along some branch. This branch represents the so-called *cube*, i.e. a conjunction of several literals. The resulting set of r cubes corresponding to cutoff branches of partitioning tree forms a set of formulas of the kind $\{G_j\}_{j=1}^{r}$ in the context of the plain partitioning strategy from [35]. The strategy described in [32] was called *Cube and Conquer*. In the recent years, Cube and Conquer SAT solvers were used to successfully solve several hard combinatorial problems related to the Ramsey theory. One of the most significant results in this area consists in solving the Boolean Pythagorean Triples Problem [33].

3.1 On Interconnection Between Plain Partitionings and Cryptographic Attacks

Another area for which the construction of SAT partitionings appears to be relevant is algebraic cryptanalysis. Below we show that a good plain partitioning of a SAT instance that encodes some cryptanalysis problem in fact may yield a cryptographic attack which is significantly better than brute force.

In the previous section we noted that a number of cryptanalysis problems can be viewed in the general context of the problem of finding preimages for functions of the kind (2). Assume that f_n is an arbitrary function of the kind (2) defined by some cipher. For example, f_n can correspond to some *keystream generator* [50] that uses an input sequence α of length n (it corresponds to either a secret key or some intermediate state of the registers) to produce a keystream fragment of length m. The cryptanalysis problem looks as follows: for a given $\gamma \in Range\ f_n$ to find $\alpha \in \{0,1\}^n$ such that $f_n(\alpha) = \gamma$. The formulated variant corresponds to the so-called *Known plaintext attack* on generator f_n [50]. Suppose that we reduced it to a problem of finding a satisfying assignment for a CNF of the kind (4). It is entirely possible that the resulting SAT instance is too hard even for the most cutting edge SAT solvers. It holds true, for example, in case of such keystream generators as Trivium [10] or Grain [31]. On the other hand, Lemma 1 states that for an arbitrary $\alpha' = (\alpha'_1, \ldots, \alpha'_n)$, $\alpha' \in \{0,1\}^n$ we can consider a CNF of the kind (3), use Unit Propagation to derive from it the corresponding $\gamma' = (\gamma'_1, \ldots, \gamma'_m)$, such that $f_n(\alpha') = \gamma'$, and check whether γ' is equal to γ. If yes, then α' is the sought key. Otherwise, we check the next α'. The described method essentially represents a variant of *brute force attack*, and its complexity is $2^n \times T_0$, where T_0 is the time required to check one key candidate. For some ciphers it is possible to find a set $B \subset X$ with the following properties:

1. $|B| = s, s < n$,
2. $2^s \cdot \widehat{T}_{\mathcal{A}} << 2^n \cdot T_0$.

Here by $\widehat{T}_{\mathcal{A}}$ we mean an upper bound on the runtime of some algorithm $\mathcal{A}$ that solves the SAT instances obtained by substituting the values of variables from B to CNF $C(f_n, \gamma)$. If a set B is found such that properties 1-2 hold for the majority of $\gamma \in Range\ f_n$, then it can be said that there exists a non-trivial *guess-and-determine attack based on guessed bits set B on function f_n*.

The guess-and-determine attacks form one of the most voluminous classes of attacks in algebraic cryptanalysis [4]. It is important to note, that the problems of inversion of functions of the kind (2) can be effectively reduced to solving systems of algebraic equations over a finite field (usually, $GF(2)$), instead of SAT. It does not change the general concept of a guess-and-determine attack.

Let B be some guessed bits set. From the point of view of the notation introduced earlier, we can associate with an arbitrary set $B = \{b_1, \ldots, b_s\}$, $B \subseteq X$ and an arbitrary $\gamma \in Range\ f_n$ a plain partitioning $P(C(f_n, \gamma), B)$ formed by the formulas of the kind

$$G(\beta) \wedge C(f_n, \gamma) \tag{5}$$

over all possible $\beta \in \{0,1\}^s$, $\beta = (\beta_1, \ldots, \beta_s)$, where

$$G(\beta) = l_{\beta_1}(b_1) \wedge \ldots \wedge l_{\beta_s}(b_s).$$

Let us apply to an arbitrary formula of the kind (5) some complete CDCL SAT solver. If the properties listed above are satisfied for a considered B then we have a non-trivial guess-and-determine attacks based on B on function f_n, which uses a SAT solver in the role of the algorithm $\mathcal{A}$. Note, that in terms of the notation from [34], formula $G(\beta)$, $\beta \in \{0,1\}^s$ in (5) is a cube over set B.

We have just described a simple method for constructing SAT partitionings: for an arbitrary set $B \subseteq X$ construct a set of formulas of the kind (5) over all possible $\beta \in \{0,1\}^{|B|}$ (hereinafter by $\{0,1\}^{|B|}$ we denote the set of all possible assignments of variables from B). It is assumed that after this one applies some complete SAT solving algorithm to CNFs (5). The above strongly correlates with a well known notion of strong backdoor sets introduced in [71].

Let C be an arbitrary CNF over X, and B, $|B| = s$, be an arbitrary subset of X. Let us use the following notation:

$$C[\beta/B] = G(\beta) \wedge C, \tag{6}$$

where $\beta = (\beta_1, \ldots, \beta_s)$, $\beta \in \{0,1\}^{|B|}$.

Definition 2 Let C be an arbitrary CNF over a set of variables X and B, $B \subseteq X$ be an arbitrary nonempty set. Let us refer to the set B as *decomposition set* for CNF C. The set of formulas of the kind (5) for all possible $\beta \in \{0,1\}^{|B|}$ is called a *SAT partitioning of C generated by decomposition set B*.

Definition 3 ([71])

Let C be an arbitrary CNF over a set of variables X, and let $\mathcal{A}$ be a polynomial-time algorithm. A nonempty set B, $B \subseteq X$, is a *strong backdoor set* for C w.r.t. algorithm $\mathcal{A}$ if for each $\beta \in \{0,1\}^{|B|}$ algorithm $\mathcal{A}$ decides formula $C[\beta/B]$.

Thus, a strong backdoor set is such a decomposition set $B \subseteq X$, that SAT for an arbitrary CNF of the kind (6) can be solved in polynomial time. It is important to note that from Lemma 1 it follows that when f_n is an arbitrary discrete function of the kind (2), then in CNFs $C(f_n)$, $C(f_n, \gamma)$ the set X^{in} is the strong backdoor set w.r.t. the Unit Propagation rule. Below let us refer to such a set as *strong Unit Propagation backdoor set (SUPBS)*.

Further we show how to relinquish the requirement that $\mathcal{A}$ must have a polynomial time complexity and employ in the role of $\mathcal{A}$ a CDCL SAT solver with finite runtime on an arbitrary input. Taking all this into account it is possible to define another subclass of decomposition sets.

Definition 4 ([66])

Assume that $\mathcal{A}$ is an arbitrary complete SAT solving algorithm, and C is an arbitrary CNF over the set X of Boolean variables. Denote by $t(\mathcal{A}, C)$ the runtime

of $\mathcal{A}$ on C. A nonempty set $B \subseteq X$ is called a *Non-Deterministic Oracle Backdoor Set* (NOBS) [66] for CNF C w.r.t $\mathcal{A}$ if

$$\sum_{\beta \in \{0,1\}^{|B|}} t(\mathcal{A}, C[\beta/B]) < t(\mathcal{A}, C).$$

In other words NOBS is such a decomposition set $B \subseteq X$ that the total runtime of $\mathcal{A}$ over all SAT instances in the SAT-partitioning $P(C, B)$ generated by B is lower than the runtime of $\mathcal{A}$ on the original CNF C.

Hereinafter we study only the SAT partitionings generated by decomposition sets in the context of Definition 2. In the next subsection we show that for SAT partitionings from this class it is possible to naturally define the measures of their effectiveness. In the role of such measures we use the estimations of the time required to solve all SAT instances from the corresponding SAT partitioning. We say that a SAT partitioning P_1 is more effective than SAT partitioning P_2 if the value of the introduced measure for P_1 is lower than that for P_2. When considering problems of inversion of cryptographic functions we can use effective SAT partitionings to construct non-trivial guess-and-determine attacks. The set which generates the corresponding SAT partitioning in that cases acts as a set of guessed bits.

To define the measures of SAT partitionings' effectiveness we employ the Monte Carlo method. We would like to note, that this term is often used without any strict formal justifications, when it basically implies the use of random sampling. Below we present the formal basis that allows one to use the Monte Carlo method in its classical form [51] to evaluate the effectiveness of SAT partitionings. For the first time the corresponding results were presented in papers [63, 62], however, similar ideas, albeit without strict justification, were employed in earlier works, e.g., [49, 22, 68].

3.2 Using Monte Carlo Method to Estimate Runtime of SAT-based Guess-and-determine Attacks

So, let us consider the inversion problem for a function of the kind (2) for a known image $\gamma \in Range\ f_n$. First reduce it to the problem of finding a satisfying assignment of CNF $C(f_n, \gamma)$. Assume that X is the set of variables from $C(f_n, \gamma)$. To solve SAT we will use a complete deterministic SAT solver $\mathcal{A}$.

Consider an arbitrary decomposition set $B \subseteq X$, $B = \{b_1, \ldots, b_s\}$. Define a uniform distribution over set $\{0,1\}^s$. With an arbitrary $\beta = (\beta_1, \ldots, \beta_s)$, randomly chosen from $\{0,1\}^s$, let us associate the value of random variable ξ which is equal to runtime of $\mathcal{A}$ on the corresponding CNF of the kind (5). Denote the spectrum of values of ξ over all possible CNFs of the kind (5) as $Sp(\xi) = \{\xi_1, \ldots, \xi_Q\}$. It is important to note that ξ has a finite expected value $E[\xi]$ and finite variance $Var(\xi)$ due to the fact that $\mathcal{A}$ has finite runtime on any CNF. The following fact was established in [62].

Theorem 1 *Let $C = C(f_n, \gamma)$ be a CNF of the kind* (4) *over a set of Boolean variables X. For an arbitrary $B \subseteq X$, $|B| = s$ consider the SAT partitioning $P(C, B)$ formed by the formulas of the kind* (5)*. Assume that $\mathcal{A}$ is a complete deterministic SAT solving algorithm, ξ is the random variable introduced above and $T(\mathcal{A}, C, B)$ is the total runtime of $\mathcal{A}$ on all SAT instances from $P(C, B)$. Then*

$$T(\mathcal{A}, C, B) = 2^{|B|} \cdot E[\xi]. \tag{7}$$

Proof Assume that all conditions of the theorem are satisfied. Consider random variable ξ and its spectrum $Sp(\xi) = \{\xi_1, \ldots, \xi_Q\}$. Note, that $Sp(\xi)$ is formed by finite positive real numbers. Let us associate with ξ the possibility space Ω, the elements of which are Boolean vectors from $\{0,1\}^s$. With each $\xi_j \in Sp(\xi)$, $j \in \{1, \ldots, Q\}$ we link a set Ω_j, $\Omega_j \subseteq \Omega$, formed by Boolean vectors $\beta = (\beta_1, \ldots, \beta_s)$ such that the runtime of $\mathcal{A}$ on each CNF of the kind (5) is ξ_j. Let us introduce the notation $\#\xi_j = |\Omega_j|$. Consider the set of numbers

$$P(\xi) = \left\{ \frac{\#\xi_1}{2^s}, \ldots, \frac{\#\xi_Q}{2^s} \right\}.$$

It is clear that the numbers from $P(\xi)$ can be viewed as the probabilities with which random variable ξ takes corresponding values from its spectrum. Thus, $P(\xi)$ is a probability distribution of random variable ξ. Taking all this into account it holds that:

$$T(\mathcal{A}, C, B) = \sum_{j=1}^{Q} \#\xi_j \cdot \xi_j = 2^s \cdot \sum_{j=1}^{Q} \frac{\#\xi_j}{2^s} \cdot \xi_j = 2^s \cdot E[\xi].$$

Thus, (7) holds. □

Formula (7) justifies the use of the Monte Carlo method for estimating the value of $T(\mathcal{A}, C, B)$. Indeed, let us denote by $\xi^l, \ldots, \xi^N$ the values of ξ observed in N independent probabilistic experiments. We assume that in each of the latter the vector β is chosen from $\{0,1\}^s$ with respect to the uniform distribution. Since $\mathcal{A}$ is a complete deterministic algorithm for solving SAT, it means that $\xi^1, \ldots, \xi^N$ can be viewed as a single observation of N independent random variables with the same distribution, expected value $E[\xi]$ and finite variance $Var(\xi)$. Let us consider a random variable

$$\frac{1}{N} \cdot \sum_{j=1}^{N} \xi^j. \tag{8}$$

We apply to (8) the Chebyshev's inequality [23]:

$$Pr\left\{ \left| \frac{1}{N} \cdot \sum_{j=1}^{N} \xi^j - E[\xi] \right| \leq \epsilon \right\} \geq 1 - \frac{Var(\xi)}{\epsilon^2 \cdot N}. \tag{9}$$

From (9) it follows that the value of $E[\xi]$ can be estimated by the values of (8) with an arbitrary tolerance $\epsilon > 0$ by increasing the number of observations N.

However, when constructing Monte Carlo estimations for functions (7) in application to practical problems, it is important to take into account several issues. The first issue is that we can construct an accurate estimation only if the variance $Var(\xi)$ is relatively small. Unfortunately, in practice it takes place quite rarely. For example, in [64] it was noted that when calculating the values of (8) for cryptanalysis of the Bivium cipher [10], the random samples of size $N < 10^4$ result in overly optimistic estimations. It happens because hard SAT instances form relatively small portion of the SAT partitioning and therefore they are often absent in samples of small sizes. This fact leads to the significant growth of sample variance with the increase of the sample size. The conclusion of [64] was that such effects are caused by the phenomenon known as heavy-tailed behavior of complete SAT solvers [27]. Another issue that has to be taken into account is that if we apply the described method to cryptographic functions, then even when $Var(\xi)$ is small, we construct the estimation of an attack runtime only for a single specific $\gamma \in Range f_n$. Nevertheless, the construction of Monte Carlo estimations for (7) and similar functions is of particular interest not only in the context of cryptanalysis, but also for the problems in which the goal is to prove the unsatisfiability of some CNF (e.g., in verification problems [42]).

Another approach to estimating the runtime of guess-and-determine attacks on functions of the kind (2) was described in [66]. Once again, consider the problem of inversion of an arbitrary function of the kind (2). However, this time our goal is to construct a guess-and-determine attack on f_n, for which the runtime estimation does not depend on particular $\gamma \in Range\ f_n$.

Assume that there is an algorithm $A(f)$ that defines f_n, a circuit $S(f_n)$, and the corresponding template CNF $C = C(f_n)$ over a set X of Boolean variables. Let us outline in X the subset $X^{in} = \{x_1, \ldots, x_n\}$ formed by the variables corresponding to the inputs of circuit $S(f_n)$, and the subset $Y = \{y_1, \ldots, y_m\}$ of variables corresponding to $S(f_n)$ outputs. The notations $C[\alpha/X^{in}]$ and $C[\gamma/Y]$ (for arbitrary $\alpha \in \{0,1\}^{|X^{in}|}$ and $\gamma \in \{0,1\}^{|Y|}$, respectively) that we use below are similar in spirit to (6).

The main distinction between the guess-and-determine attack described further from the one outlined above consists in the following: we limit the runtime of SAT solver $\mathcal{A}$ by some τ on CNFs from a SAT partitioning generated by B. If the runtime of $\mathcal{A}$ on an arbitrary CNF from the partitioning exceeded τ then $\mathcal{A}$ is interrupted.

The notions we introduce further are based on the property formulated in Lemma 1. Recall that according to this property, for an arbitrary $\alpha \in \{0,1\}^{|X^{in}|}$ the application of Unit Propagation to CNF $C[\alpha/X^{in}]$ results in derivation of values of all variables from X including the value of f_n. For an arbitrary set $B \subseteq X$ let us refer to the values of its variables derived by Unit Propagation from CNF (3) as to an *assignment induced by* α.

Define a uniform distribution over $\{0,1\}^n$. Fix an arbitrary $B \subseteq X \backslash Y$. With a randomly chosen $\alpha \in \{0,1\}^n$ associate the assignments $\beta(\alpha)$ and $\gamma(\alpha) = f_n(\alpha)$ of variables from B and Y, respectively, which were induced by α from CNF (3). Consider CNF $C[\gamma(\alpha)/Y, \beta(\alpha)/B]$ where $C = C(f_n)$. Let us denote by $t(\mathcal{A}, C, B, \alpha)$ the runtime of $\mathcal{A}$ on the input $C[\gamma(\alpha)/Y, \beta(\alpha)/B]$. Assume that the value $\tau > 0$ is fixed to some constant. Then consider the following value:

$$p_B(\tau) = \frac{\#\{\alpha \in \{0,1\}^n : t(\mathcal{A}, C, B, \alpha) \leq \tau\}}{2^n} \tag{10}$$

The numerator of (10) represents the number of such $\alpha \in \{0,1\}^n$ for which CNF $C[\gamma(\alpha)/Y, \beta(\alpha)/B]$ is decided by $\mathcal{A}$ in time $\leq \tau$. The denominator of (10) is the number of all α. Therefore, (10) is essentially the probability of the following event: that a randomly selected $\alpha \in \{0,1\}^n$ induces such assignments $\beta(\alpha)$ and $\gamma(\alpha)$ that $\mathcal{A}$ decides $C[\gamma(\alpha)/Y, \beta(\alpha)/B]$ in time $\leq \tau$.

Definition 5 ([66])

An arbitrary nonempty decomposition set B, $B \subseteq X \backslash Y$, $|B| = s$, with properties described above is called an *Inverse Backdoor Set* (IBS) with parameters $(s, \tau, p_B(\tau))$ for $C(f_n)$ w.r.t. algorithm $\mathcal{A}$.

In [66] there was introduced a new class of guess-and-determine attacks. They are based on the following two definitions.

Definition 6 (IBS-based elementary guess-and-determine attack, [66])

Consider the inversion problem for a function f_n of the kind (2). Assume that we have an arbitrary $\gamma \in Range\ f_n$ that corresponds to some $\alpha \in \{0,1\}^n$ (i.e. $\gamma = f_n(\alpha)$).

1. Let B be an IBS with parameters $(s, \tau, p_B(\tau))$ and $\beta \in \{0,1\}^s$ be an assignment to variables of B.
2. Construct CNF $C[\gamma/Y, \beta/B]$ and run SAT solver $\mathcal{A}$ on it.
3. If the runtime of $\mathcal{A}$ on this SAT instance exceeds τ, interrupt the solving process and move to another $\beta \in \{0,1\}^{|B|}$.
4. For $\beta = \beta(\alpha)$ algorithm $\mathcal{A}$ will find a satisfying assignment for $C[\gamma/Y, \beta/B]$ in time $\leq \tau$ with probability $p_B(\tau)$. This means that in this case $\mathcal{A}$ will compute α s.t. $f_n(\alpha) = \gamma$.

It is possible that the analysis of some γ leads to no result because for each formula $C[\gamma/Y, \beta/B], \beta \in \{0,1\}^{|B|}$ either the runtime of $\mathcal{A}$ exceeded τ or the unsatisfiability was proved. In that case due to the cryptographic context we can consider another element from $Range\ f_n$ that is different from γ. For example, if f_n is some keystream generator then we can consider a fragment of keystream of size m that follows after γ. Thus, we have the following iterative guess-and-determine attack.

Definition 7 (IBS-based guess-and-determine attack, [66])

Consider the inversion problem for a function f_n of the kind (2).

1. Let $\gamma^1, \ldots, \gamma^r$ be observed outputs of function f_n. These outputs correspond to inputs $\alpha^1, \ldots, \alpha^r$.
2. Let B be some IBS with parameters $(s, \tau, p_B(\tau))$, $p_B(\tau) > 0$.
3. A guess-and-determine attack based on IBS B consists in successive application of elementary attack, as described in Definition 6, to outputs $\gamma^1, \ldots, \gamma^r$.
4. The attack is said to be *successful* if for at least one $j \in \{1, \ldots, r\}$ the corresponding inversion problem for f_n is solved.

For an arbitrary function of the kind (2) by the runtime of a corresponding guess-and-determine attack we mean the time required by this attack until at least one image of function f_n is inverted.

Definition 8 Let P_r^* be the probability of the event that a guess-and-determine attack from Definition 7 is successful. Let us say that this attack is *statistically significant* if $P_r^* \geq 0.95$.

Let us denote the runtime of a statistically significant guess-and-determine attack as $T(\mathcal{A}, C, B, \tau)$, where $C = C(f_n)$. The following result was implicitly presented in [66].

Theorem 2 *Consider an inversion problem for a function f_n of the kind* (2)*. Assume that in the context of this problem B is an arbitrary IBS with parameters $(s, \tau, p_B(\tau))$, $p_B(\tau) > 0$. Then there exists a statistically significant guess-and-determine attack based on IBS B with runtime*

$$T(\mathcal{A}, C, B, \tau) = 2^s \cdot \tau \cdot \left\lceil \frac{3}{p_B(\tau)} \right\rceil. \tag{11}$$

Proof Assume that the conditions of the theorem are satisfied. In accordance with Definition 6 the $p_B(\tau)$ is the probability that $\mathcal{A}$ will invert an arbitrary $\gamma \in Range\ f_n$ after traversing all the $\beta \in \{0,1\}^{|B|}$, and spending on each β the time $\leq \tau$. If we consider the inputs $\alpha_1, \ldots, \alpha_r$ independently chosen from $\{0,1\}^n$ and the corresponding outputs $\gamma_1, \ldots, \gamma_r$ of f_n then it is easy to see that

$$P_r^* = 1 - (1 - p_B(\tau))^r.$$

Assume that $r = \left\lceil \frac{3}{p_B(\tau)} \right\rceil$ and consider the value

$$(1 - p_B(\tau))^{\left\lceil \frac{3}{p_B(\tau)} \right\rceil} \leq (1 - p_B(\tau))^{\frac{3}{p_B(\tau)}}.$$

The value in the right hand side of the latter inequality monotonically increases with the decrease of $p_B(\tau)$ and does not exceed $e^{-3} \approx 0.04978$. □

We would like to additionally note that the value $p_B(\tau)$ does not depend on a specific output of f_n, and thus (11) characterizes the runtime of the described attack for $\left\lceil \frac{3}{p_B(\tau)} \right\rceil$ arbitrary outputs of f_n induced by random inputs.

Similarly to function (7) the values of function (11) can be estimated using the Monte Carlo method. In more detail, we can use this method to estimate the probability $p_B(\tau)$. For this purpose let us associate with the possibility space $\Omega = \{0,1\}^n$ the random variable ζ that takes values from the set $\{0,1\}$. If for an arbitrary $\alpha \in \{0,1\}^n$ the algorithm $\mathcal{A}$ decides the satisfiability of CNF $C[\gamma(\alpha)/Y, \beta(\alpha)/B]$ in time $\leq \tau$, then we say that $\zeta = 1$, otherwise $\zeta = 0$. It is clear that ζ is a random variable with the spectrum $Sp(\zeta) = \{1, 0\}$ and probability distribution $\{p_B(\tau), 1 - p_B(\tau)\}$. Thus, ζ is the Bernoulli random variable with success probability $p_B(\tau)$. Therefore, $E[\zeta] = p_B(\tau), Var(\zeta) = p_B(\tau) \cdot (1 - p_B(\tau))$. Taking this into account let us apply the Chebyshev's inequality to produce the following statement (similarly to (9)):

$$Pr\left\{\left|\frac{1}{N}\cdot\sum_{j=1}^{N}\zeta^j - p_B(\tau)\right| \leq \epsilon\right\} \geq 1 - \frac{1}{4\cdot\epsilon^2\cdot N}. \quad (12)$$

The right hand side of (12) makes use of the fact that the function $x\cdot(1-x)$ achieves its maximum over $[0,1]$ in $x = 0.5$.

Note, that (12) looks beneficially compared to (9) because it lacks the unknown variance. Thus, in the IBS-based attacks we can guarantee the precision of the constructed estimations of their runtime by increasing the size N of a random sample. From the other hand, the method presented in [63, 62] makes it possible to construct runtime estimations for arbitrary SAT partitionings, including that for unsatisfiable CNFs, while IBS-based approach can not be applied to the latter.

4 Practical Aspects of Evaluating Effectiveness of SAT Partitionings

In this section we describe how we can represent the problem of finding an effective SAT partitioning as a black-box optimization problem. We also consider several implementation level details that make it possible to sometimes substantially improve the performance of the proposed approach in practice.

First, let us briefly recapture the results of the previous section. It was shown that given a CNF C over a set of Boolean variables X and some subset $B \subseteq X$ it is possible to use this subset to partition C into a family of (usually) easier subproblems. Due to peculiar features of SAT, it is entirely possible that for a given SAT solving algorithm $\mathcal{A}$ its runtime on an original problem can be significantly higher than its total runtime on all subproblems from a constructed SAT partitioning [40]. In the previous section we introduced two main ways for evaluating the effectiveness of SAT partitionings. The functions (7) and (11) represent the essence of these approaches. Regardless of the approach, we want to find a SAT partitioning that has the best effectiveness.

In practice, it is infeasible to compute the exact values of (7) and (11) because it is equivalent to solving the original problem. However, as it was noted above, the Monte Carlo method makes it possible to construct probabilistic estimations of (7) and (11) that can often be computed in realistic time. The corresponding functions for (7) and (11), respectively, look as follows:

$$\Phi_{\mathcal{A},C,N}(B) = \frac{2^s}{N}\cdot\sum_{j=1}^{N}\xi^j, \quad (13)$$

$$\Psi_{\mathcal{A},C,N,\tau}(B) = 2^s\cdot\tau\cdot\frac{3}{\sum_{j=1}^{N}\zeta^j}. \quad (14)$$

It should be noted that for (14) if the value in the denominator is equal to zero, then we assume that the function's value is $+\infty$. Also, $C = C(f_n, \gamma)$ in (13) and $C = C(f_n)$ in (14).

Let us consider functions (13) and (14) in more detail. Note, that when computing the values of (13) and (14) one needs to generate a random sample of size N, observe N values of a corresponding random variable and then perform simple arithmetic operations. The N observations of random variable in case of (13) mean measuring the runtime of a SAT solving algorithm $\mathcal{A}$ on the N problems forming the sample. For (14) it is necessary to not only measure the runtime of $\mathcal{A}$ but to also interrupt it once the time limit τ is exceeded. It means that the functions (13) and (14) are not defined analytically and thus can be viewed as black-box functions. Therefore, taking into account all of the above we can move from the problem of finding a SAT partitioning with the best value of (7) or (11) to the problem of finding a set B that yields a minimum of the function (13) or (14), respectively, over some finite search space (ideally, over 2^X) for fixed $\mathcal{A}, C, N, \tau$. Thus we have the formulation of the problem of finding an effective SAT partitioning in form of the Black-box optimization problem and can apply to its solving the corresponding methods.

4.1 Narrowing Search Space to SUPBS

As it is the case for all optimization algorithms, if there is a possibility to reduce the search space – it must be used. Fortunately, in the case of functions (13) and (14), Lemma 1 provides just the opportunity to do so, at least for major classes of problems. Essentially, since from Lemma 1 it follows that the variables from X^{in} form a SUPBS (see Sect. 3.1) for both $C(f_n)$ and $C(f_n, \gamma)$, then they can be viewed as much more important than the other variables in C. This fact fits nicely with the cryptographic specifics of the problems considered in later sections. Thus, we can narrow down the search space as follows: we choose a set B only from the set of subsets of a set X^{in}. Note, that from Lemma 1 it follows that for $B = X^{in}$ the values of (13) and (14) can be computed effectively in polynomial time in the size of binary description of CNF C.

So, assume that the set of possible alternatives of B is $2^{X^{in}}$, $X^{in} = \{x_1, \ldots, x_n\}$. Then we can represent any $B \subseteq X^{in}$ by a Boolean vector $\lambda = (\lambda_1, \ldots, \lambda_n)$, assuming that $\lambda_i = 1$, $i \in \{1, \ldots, n\}$ if and only if $x_i \in B$. Otherwise, $\lambda_i = 0$. Taking this into account, (13) and (14) are pseudo-Boolean functions [8]:

$$\Phi_{\mathcal{A},C,N} : \{0,1\}^n \to \mathbb{R}, \Psi_{\mathcal{A},C,N,\tau} : \{0,1\}^n \to \mathbb{R}.$$

For an arbitrary $\lambda \in \{0,1\}^n$ the value of, say, $\Phi_{\mathcal{A},C,N}(\lambda)$ is computed as follows: first we construct a set B corresponding to λ. Then for this B we consider N independent observations of random variable $\xi : \xi^1, \ldots, \xi^N$ and compute (8). The value of $\Psi_{\mathcal{A},C,N,\tau}(\lambda)$ is computed in a similar fashion according to (14).

4.2 Applications of Incremental SAT Solving

It is quite clear that when computing values of (13) it is necessary to solve many similar SAT instances. As it happens, there is a special SAT solving technique that makes it possible to sometimes increase the performance of SAT solvers on such tasks. It is called *incremental SAT solving* and was described, for example, in [21]. Below let us briefly recall one of its variants that we can use in practice when computing the function (13).

Consider SAT for an arbitrary CNF C over the set X of Boolean variables. Let $B = \{b_1, \ldots, b_s\}$, $B \subseteq X$ be a decomposition set that generates SAT partitioning $P(C, B)$. Assume that there is a sequence $\rho^1, \ldots, \rho^k$, $k \geq 2$, $\rho^i = (\beta^i_1, \ldots, \beta^i_s)$, $\rho^i \in \{0,1\}^{|B|}, i = 1, \ldots, k$ of assignments of variables from B, and our goal is to solve SAT for CNFs

$$\begin{array}{c} G(\rho^1) \ \wedge \ C \\ \ldots \\ G(\rho^k) \ \wedge \ C \end{array}$$

where $G(\rho^i) = l_{\beta^i_1}(b_1) \wedge \ldots \wedge l_{\beta^i_s}(b_s)$. The essence of the incremental SAT solving technique consists in the augmentation of the standard CDCL algorithm that allows it to process a sequence of CNFs $G(\rho^1) \wedge C, G(\rho^2) \wedge C, \ldots, G(\rho^k) \wedge C$, in a single launch of SAT solver. For this purpose the literals of the kind $l_{\beta^i_1}(b_1), \ldots, l_{\beta^i_s}(b_s)$ are not added to C as unit clauses. Instead, the SAT solver uses them as the so-called *assumptions* [21]. It means that it treats $\beta^i_1, \ldots, \beta^i_s$ as guessed values of the corresponding variables. Thus, it assigns the value β^i_1 to b_1 the same way it would do it when following an arbitrary path in the search tree. As a result of this trick, any learnt clause that was derived while processing $G(\rho^i) \wedge C$ can be reused when solving $G(\rho^j) \wedge C$, $i \neq j$ or even C. Observe that $G(\rho^i)$ naturally corresponds to the elements of a random sample used when computing (13). It is currently unclear whether the incremental processing can be naturally applied to computing (14).

In practice the incremental SAT solving has several specific features that should be taken into account. As it was already noted, the learnt clauses generated for one assignment of variables from B can be reused while solving SAT for another. It means that we can solve not individual instances, but blocks of k instances per single launch of the SAT solver, $k > 1$. The value of k has a significant impact on the performance of the incremental SAT solver and is usually chosen to be relatively small, e.g., $10 \leq k \leq 100$. Since one of the benefits of incremental SAT solving consists in not having to initialize the solver for each particular instance, it means that the effect tends to be better when individual instances are relatively easy and can be solved in tens of seconds at most. Otherwise, it is often the case that the state-of-the-art preprocessing and inprocessing methods yield better results when solving $G(\rho^i) \wedge C$ individually.

4.3 Finding Partitionings via Incremental SAT

As it follows from the previous subsection, using incremental SAT we can decrease the time required by a solver to compute $\Phi_{\mathcal{A},C,N}(B)$ especially for larger N. It appears to be extremely worthwhile in the case when we use B to solve SAT for C, i.e. when we need to process all $N{=}2^{|B|}$ subproblems from a partitioning.

The goal of the function introduced in [75] is to evaluate the effectiveness of SAT partitionings using state-of-the-art incremental SAT solvers. Consider a Boolean hypercube $\{0,1\}^{|B|}$, $|B| = s$. With an arbitrary $i \in \{1,\ldots,2^s\}$ associate an s-bit vector ρ^i which represents the number $i-1$ in binary. Consider $\{0,1\}^{|B|}$ as a set of vectors of the kind ρ^i over all possible $i \in \{1,\ldots,2^s\}$. Fix an arbitrary $k : 2 \le k < 2^s$ and split $\{0,1\}^{|B|}$ into distinct subsets of the following kind:

$$\begin{gathered} I_1 = \{\rho^1,\ldots,\rho^k\}, I_2 = \{\rho^{k+1},\ldots,\rho^{2k}\},\ldots, \\ I_{U-1} = \{\rho^{(U-2)\times k+1},\ldots,\rho^{(U-1)\times k}\}, I_U = \{\rho^{(U-1)\times k+1},\ldots,\rho^{2^s}\} \end{gathered} \tag{15}$$

In (15), $U = \lceil \frac{2^s}{k} \rceil$, $2^s = k \cdot \left(\lceil \frac{2^s}{k} \rceil - 1 \right) + |I_U|$, $|I_U| \in \{1,\ldots,k\}$. Further let us use the following notation.

$$I^B = \{I_1,\ldots,I_U\}$$

We will refer to I^B as to an *interval partition of rank k* of hypercube $\{0,1\}^{|B|}$.

The basic idea of the approach presented in [75] is to allow a SAT solver to process all vectors from an arbitrary interval $I \in I^B$ incrementally when solving the subproblems from a SAT partitioning $P(C,B)$.

It is possible to define the effectiveness of $P(C,B)$ with respect to a complete deterministic incremental SAT solver $\mathcal{A}$ in a similar way as it was done for (7). Associate with $\{0,1\}^{|B|}$ and its interval partition I^B of the kind (15) a value η, the spectrum of which is formed by all possible runtimes of $\mathcal{A}$ on intervals from I^B. The possibility space associated with η is I^B. Define a uniform distribution over I^B. Let us denote by $T(\mathcal{A},C,B,I^B)$ the time required to solve all problems from a SAT partitioning $P(C,B)$ by incremental SAT solver $\mathcal{A}$ that employs the interval partition I^B of the hypercube $\{0,1\}^{|B|}$. Then, similar to (7) it holds that

$$T(\mathcal{A},C,B,I^B) = U \cdot E[\eta]. \tag{16}$$

Now let us define a pseudo-Boolean function the values of which will estimate (16) for various B and the corresponding interval partitions (15) with fixed $\mathcal{A}$, C, and N:

$$\Upsilon_{\mathcal{A},C,N}(B,I^B) = \frac{U}{N} \cdot \sum_{j=1}^{N} \eta^j. \tag{17}$$

In (17) by η^j, $j \in \{1,\ldots,N\}$ we denote the values of random variable η observed in N independent observations. In each experiment an interval I is first randomly chosen from I^B according to the uniform distribution. Then an incremental SAT

solver $\mathcal{A}$ is launched on I. The observed value of η is the total runtime of $\mathcal{A}$ over all problems from I.

5 Employed Optimization Algorithms

The objective functions (13), (14), and (17) proposed in two previous sections are pseudo-Boolean black-box costly stochastic functions. Therefore they can be minimized by pseudo-Boolean black-box optimization algorithms based on direct calculations (see, e.g., [1, 41, 45, 57]). In this section several such algorithms are described that are further used in Sect. 6. The list is presented below.

1. steepest ascent hill climbing (SAHC);
2. first-choice hill climbing (FCHC);
3. first-choice hill climbing with variables-based jump (FCHCVJ);
4. tabu search (TS);
5. tabu search with activity-based escape (TSAE);
6. simulated annealing (SA);
7. (1+1) evolutionary algorithm ((1+1)-EA).
8. genetic algorithm (GA);

Some features of the considered algorithms are outlined in Table 1. Note that all three objective functions are extremely costly, that is why in almost all employed optimization algorithms (except SAHC and FCHC) a *tabu list* is used to prevent calculating an objective function's value at any point more than once. If not mentioned otherwise, it is implied that the tabu list stores all processed points. In *trajectory-based* algorithms a successor solution is sought in a neighborhood of a current solution. *Escaping local minima* means that a trajectory-based algorithm uses a specific heuristic to move out of points with neighborhoods in which it could not improve a function's value.

Table 1: Some features of the considered optimization algorithms.

Algorithm	Stochastic	Tabu lists	Trajectory-based	Escaping local minima
SAHC	No	No	Yes	No
FCHC	Yes	No	Yes	No
FCHCVJ	Yes	Yes	Yes	Yes
TS	No	Yes	Yes	Yes
TSAE	No	Yes	Yes	Yes
SA	Yes	Yes	Yes	Yes
(1+1)-EA	Yes	Yes	No	-
GA	Yes	Yes	No	-

Since the functions (13), (14), and (17) are pseudo-Boolean, it means that they take a Boolean vector $\lambda = (\lambda_1, \ldots, \lambda_n)$ as an input. Such a vector is considered

as a point in the search space $\{0,1\}^n$. Thus, it is quite natural to operate with the Hamming distance [30] between points to form a neighborhood. To be more specific, we assume that a neighborhood of a point λ contains all points that are at Hamming distance at most H from λ. Hereinafter we refer to a point for which an objective function value has already been calculated as *processed point*. Similarly, a neighborhood is called *processed* if all its points are processed, otherwise it is called *unprocessed*.

Let us describe the mentioned optimization algorithms in more detail. In the corresponding pseudocodes, g stands for an objective function ((13), (14), or (17)), g_{best} and λ_{best} – for the *best known value* (BKV) of the objective function and the corresponding point, respectively. Every algorithm is given a starting point λ_{start}. It goes without saying that several parameters are common for all functions: CNF C, SAT solving algorithm $\mathcal{A}$, random sample size N (see Sect. 3). Some of the objective functions require additional parameters, but for brevity we do not mention them in pseudocodes. Instead, it is implied that all the required ones are given.

The *steepest ascent hill climbing* algorithm (see, e.g., [59]) calculates objective function values for all points from a current neighborhood and chooses the best candidate from them (in our case – the one with the lowest objective function value). If the best candidate is better than the current λ_{best}, then both λ_{best} and g_{best} are updated. If all points from the neighborhood are worse than the current λ_{best}, then a local minimum is reached and the algorithm stops. The pseudocode is shown in Alg. 1. For a given point λ, the function `getRandOrdNeighb(`λ`,H)` returns the list of randomly ordered points that are at Hamming distance at most H from λ. Note, that for this algorithm any ordering can be used, but random ordering is required for the first-choice hill climbing that is described further.

Input: Starting point λ_{start}, Hamming distance H, time limit t

1 $\langle \lambda_{best}, g_{best} \rangle \leftarrow \langle \lambda_{start}, g(\lambda_{start}) \rangle$
2 **repeat**
3 BestValueUpdated $\leftarrow$ false
4 CurNeigh $\leftarrow$ `getRandOrdNeighb`(λ_{best}, H)
5 **for** `each` λ `in` CurNeigh **do**
6 NewFuncValue $\leftarrow g(\lambda)$
7 **if** NewFuncValue $< g_{best}$ **then**
8 $\langle \lambda_{best}, g_{best} \rangle \leftarrow \langle \lambda,$ NewFuncValue$\rangle$
9 BestValueUpdated $\leftarrow$ true
10 **until** `timeExceeded(`t`)` `or not` BestValueUpdated
11 **return** $\langle \lambda_{best}, g_{best} \rangle$

Algorithm 1: Steepest ascent hill climbing

In opposite to steepest ascent hill climbing, the *first-choice hill climbing* algorithm (see, e.g., [59]) immediately stops processing the current neighborhood as soon as λ_{best} is updated. Thus the pseudocode of first-choice hill climbing can be easily made by adding the line *Break* after line number 9 in Alg. 1. Since both steepest ascent hill climbing and first-choice hill climbing do not have heuristics for escaping local minima, they are not used in experiments described in Sect. 6. However, the

next four algorithms that we actually use for this purpose are based on either SAHC or FCHC.

Input: Starting point λ_{start}, Hamming distance H, time limit t

1 $\langle \lambda_{best}, g_{best} \rangle \leftarrow \langle \lambda_{start}, g(\lambda_{start}) \rangle$
2 $\lambda_{center} \leftarrow \lambda_{start}$
3 TabuList $\leftarrow \{\lambda_{start}\}$
4 **repeat**
5 NonTabuPoints $\leftarrow$ `getNonTabuNeighbors`(λ_{center}, H, TabuList)
6 **if** NonTabuPoints = $\emptyset$ **then**
7 `Break`
8 $g_{candidate} \leftarrow \infty$
9 **for** `each` λ `in` NonTabuPoints **do**
10 NewFuncValue $\leftarrow g(\lambda)$
11 **if** NewFuncValue $< g_{candidate}$ **then**
12 $\langle \lambda_{candidate}, g_{candidate} \rangle \leftarrow \langle \lambda$, NewFuncValue$\rangle$
13 **if** $g_{candidate} < g_{best}$ **then**
14 $\langle \lambda_{best}, g_{best} \rangle \leftarrow \langle \lambda_{candidate}, g_{candidate} \rangle$
15 `updateTabuList`($\lambda_{candidate}$, TabuList)
16 $\lambda_{center} \leftarrow \lambda_{candidate}$
17 **until** `timeExceeded`(t)
18 **return** $\langle \lambda_{best}, g_{best} \rangle$

Algorithm 2: The tabu search algorithm

Tabu search was proposed in [26]. We implemented a tabu search algorithm on the basis of steepest ascent hill climbing. A tabu list is used where the last 1000 best points from the processed neighborhoods are stored. In opposite to first-choice hill climbing and steepest ascent hill climbing, if a local minimum is reached, the algorithm does not stop. Instead, the best point from the neighborhood is chosen, and the processing of its neighborhood is started. Alg. 2 shows the corresponding pseudocode. The function `getNonTabuNeighbors(`λ`,H,TabuList)` returns randomly ordered points from the neighborhood of λ with Hamming distance at most H that are not in the tabu list. The function `updateTabuList()` adds a given point to the top of the tabu list. If the tabu list is full, the last point is removed from it before adding a new one. It means that repeated calculations are possible in this algorithm. By $\lambda_{candidate}$ and $g_{candidate}$ we denote the best non-tabu point from the current neighborhood and the objective function value for it, respectively.

Tabu search with activity-based escape was proposed in [62]. Unlike the tabu search algorithm described above, it uses another heuristic for escaping local minima, and stores all processed points. More specifically, the information about processed points is stored in two tabu lists L_1 and L_2. L_1 contains processed points with processed neighborhoods, while L_2 contains processed points with unprocessed neighborhoods. In particular, each point in L_2 is stored along with a Boolean vector that specifies which points from the corresponding neighborhood have not been processed yet. The pseudocode is presented in Alg. 3. The list of points that have not been processed so far according to both tabu lists is formed by the function `getNonTabuNeighbors2`. The function `markPointInTabuLists`(λ, L_1, L_2) adds

the point λ to L_2 and then marks λ as processed in all neighborhoods of points from L_2 that contain λ. If as a result the neighborhood of some point λ becomes processed, the point is removed from L_2 and is added to L_1. If all points in the neighborhood of λ_{center} have been processed but g_{best} has not been improved, then some point from L_2 is chosen as λ_{center} to escape the reached local minimum. It is done via the function `getNewCenter(`L_2`)` that chooses from L_2 a point with the largest total conflict activity of Boolean variables from the corresponding decomposition set. This activity is taken from a CDCL-based [46] SAT solving algorithm $\mathcal{A}$.

Input: Starting point λ_{start}, Hamming distance H, time limit t

$\langle \lambda_{best}, g_{best} \rangle \leftarrow \langle \lambda_{start}, g(\lambda_{start}) \rangle$
$\lambda_{center} \leftarrow \lambda_{start}$
$L_1 \leftarrow \emptyset$
$L_2 \leftarrow \{\lambda_{start}\}$
repeat
  BestValueUpdated ← false
  NonTabuPoints ← `getNonTabuNeighbors2(`$\lambda_{center}, H, L_1, L_2$`)`
  for `each` λ `in` NonTabuPoints **do**
    NewFuncValue ← $g(\lambda)$
    `markPointInTabuLists(`λ, L_1, L_2`)` `// update tabu lists`
    if NewFuncValue < g_{best} **then**
      $\langle \lambda_{best}, g_{best} \rangle \leftarrow \langle \lambda$, NewFuncValue$\rangle$
      BestValueUpdated ← true
  if BestValueUpdated **then**
    $\lambda_{center} \leftarrow \lambda_{best}$
  else
    $\lambda_{center} \leftarrow$ `getNewCenter(`L_2`)`
until `timeExceeded(`t`)`
return $\langle \lambda_{best}, g_{best} \rangle$

Algorithm 3: Tabu search with activity-based escape

First-choice hill climbing with variables-based jump was proposed in [40, 74]. It is a first-choice hill climbing improved by a memory-based heuristic for escaping local minima. This heuristic employs two arrays of counters: $U^1, |U^1| = n$ and $U^2, |U^2| = n$. All elements of both arrays are initialized with zeros. If an objective function is calculated in a point λ, then for all i such that $\lambda_i = 1$ the counter U^1_i is increased by one. The second vector of counters is updated in the same manner, but only for such points in which g_{best} has been updated. If a local minimum is reached in a neighborhood of a current λ_{best}, then a new starting point is constructed by changing $2m$ elements of λ_{best} from 0 to 1. The elements to flip are picked as follows: m of them correspond to m lowest values in U^1 and the other m to the m highest values in U^2. The motivation here is to move to a point that simultaneously contains the variables that often resulted in an improvement of function's value and the variables that were rarely added in the process of the search so far to balance the exploitation and exploration. In all experiments described in Sect. 6, m was equal to 4. The pseudocode is shown in Alg. 4. Functions `updateFirstCounter()` and `updateSecondCounter()` update

U^1 and U^2, respectively. The function `getJumpPoint(`λ`,m)` constructs a new point by changing $2m$ elements of a given point λ from 0 to 1 as it was described above.

Input: Starting point λ_{start}, escape parameter m, Hamming distance H, time limit t

$\langle \lambda_{best}, g_{best} \rangle \leftarrow \langle \lambda_{start}, g(\lambda_{start}) \rangle$
$\lambda_{center} \leftarrow \lambda_{start}$
TabuList $\leftarrow \{\lambda_{start}\}$
$U^1 = U^2 = \mathbf{0}$ `// initialize with zero vectors`
repeat
  NonTabuPoints $\leftarrow$ `getNonTabuNeighbors(`λ_{center}, H, TabuList`)`
  BestValueUpdated $\leftarrow$ false
  for `each` λ `in` NonTabuPoints **do**
    NewFuncValue $\leftarrow g(\lambda)$
    `addPointTabuList(`λ, TabuList`)`
    `updateFirstCounter(`λ, U^1`)`
    if NewFuncValue $< g_{best}$ **then**
      $\langle \lambda_{best}, g_{best} \rangle \leftarrow \langle \lambda$, NewFuncValue$\rangle$
      `updateSecondCounter(`λ_{best}, U^2`)`
      BestValueUpdated $\leftarrow$ true
      `Break`
  if BestValueUpdated = true **then**
    $\lambda_{center} \leftarrow \lambda_{best}$
  else
    $\lambda_{center} \leftarrow$ `getJumpPoint(`λ_{best}, m`)`
until `timeExceeded(`t`)`
return $\langle \lambda_{best}, g_{best} \rangle$

Algorithm 4: First-choice hill climbing with variables-based jump

The simulated annealing algorithm was proposed in [39]. Below we describe its variant from [62] aimed at minimizing objective functions of the considered type. It is based on steepest ascent hill climbing, A point λ from a current neighborhood becomes λ_{center} with the probability

$$Pr\{\lambda \to \lambda_{center}\} = \begin{cases} 1, & if\ g(\lambda) < g(\lambda_{best}) \\ \exp(-\frac{g(\lambda)-g(\lambda_{best}))}{T}), & if\ g(\lambda) \geq g(\lambda_{best}) \end{cases} \quad (18)$$

Here T corresponds to the "temperature of the environment" [39]. First, T is assigned a large value T_0. At each calculation of an objective function, T is decreased: $T = Q \cdot T$, where $Q \in (0, 1)$. Algorithm stops when T drops below a threshold value T_{thresh}. In [62] the following values were used: $T_0 = \frac{g(\lambda_{start})}{10}$, $Q = 0.9$, $T_{thresh} = 20$. The pseudocode is shown in Alg. 5. The function `pointAccepted(`$g(\lambda), g_{best}, T$`)` employs formula (18) to test if g becomes g_{best}.

Input: Starting point λ_{start}, Hamming distance H, time limit t, temperature decrease parameter Q, temperature threshold T_{thresh}

$\langle \lambda_{best}, g_{best} \rangle \leftarrow \langle \lambda_{start}, g(\lambda_{start}) \rangle$
$T = \frac{g(\lambda_{start})}{10}$
TabuList $\leftarrow \{\lambda_{start}\}$
repeat
  NonTabuPoints $\leftarrow$ `getNonTabuNeighbors`$(\lambda_{best}, H,$ TabuList$)$
  if NonTabuPoints $= \emptyset$ **then**
    `Break`
  for `each` λ `in` NonTabuPoints **do**
    NewFuncValue $\leftarrow g(\lambda)$
    `addPointTabuList`$(\lambda,$ TabuList$)$
    if `pointAccepted` (NewFuncValue, g_{best}, T) **then**
      $\langle \lambda_{best}, g_{best} \rangle \leftarrow \langle \lambda,$ NewFuncValue$\rangle$
    $T = T \cdot Q$
until `timeExceeded`(t) `or` $T < T_{thresh}$
return $\langle \lambda_{best}, g_{best} \rangle$

Algorithm 5: The simulated annealing algorithm

The *(1+1) evolutionary* algorithm is described, e.g., in [52]. Below we employ this algorithm in the form that was used in [55] for minimization of the objective function (15). Unlike the original algorithm, it starts from a given point. Alg. 6 shows the pseudocode. The function `mutate`$(\lambda, \frac{1}{n})$ flips independently each element of a given point λ (which in turn is a Boolean vector of size n) with probability $\frac{1}{n}$. This function is run until a non-tabu point is obtained.

Input: Starting point λ_{start}, time limit t

$\langle \lambda_{best}, g_{best} \rangle \leftarrow \langle \lambda_{start}, g(\lambda_{start}) \rangle$
TabuList $\leftarrow \{\lambda_{start}\}$
repeat
  repeat
    $\lambda_{flipped} \leftarrow$ `mutate`$(\lambda_{best}, \frac{1}{n})$
  until $\lambda_{flipped}$ `not in` TabuList
  NewFuncValue $\leftarrow g(\lambda_{flipped})$
  `addPointTabuList`$(\lambda,$ TabuList$)$
  if NewFuncValue $< g_{best}$ **then**
    $\langle \lambda_{best}, g_{best} \rangle \leftarrow \langle \lambda_{flipped},$ NewFuncValue$\rangle$
until `timeExceeded`(t)
return $\langle \lambda_{best}, g_{best} \rangle$

Algorithm 6: The (1+1) evolutionary algorithm

Finally, we present a variant of *genetic algorithm* proposed in [55]. Each point $\lambda \in \{0,1\}^n$ is considered as an *individual* (see, e.g., [45]), the fitness of which is determined by the value of an objective function in the corresponding point. Alg. 7 shows the pseudocode. The sizes of all populations are fixed to K. The starting population is initialized via the function `initPopulation`(λ_{start}, K) that constructs K replicas of λ_{start}. The key function of the genetic algorithm is the one used to move from the

current population to the next. In Alg. 7 it is `getNextPopulation`(P_{curr}, E, M, R). Assume that $P_{curr} = \{I_1, \ldots, I_K\}$ and $P_{next}, |P_{next}| = K$ are a current and a new population, respectively. P_{next} is formed by $E + M + R = K$ points that are chosen as follows. First E are chosen as the top E individuals with the best values of the objective function from P_{curr}. It is done in accordance with the so-called *elitism* principle (see, e.g., [45]). Next, we compute auxiliary values

$$p_j = \frac{\frac{1}{g(I_j)}}{\sum_{u=1}^{K} \frac{1}{g(I_u)}}, j = 1, \ldots, K$$

and use them to form probability distribution $D_{curr} = \{p_1, \ldots, p_K\}$. Then M individuals are chosen randomly from P_{curr} according to D_{curr}, undergo standard (1+1)-EA mutation and go into P_{next}. Finally, R pairs of individuals are chosen from P_{curr} according to D_{curr} to perform the standard two-point crossover. The result of crossover goes to P_{next}. The situation when the algorithm fails to improve the BKV during one generation is called *stagnation*. If the number of stagnations exceeds a given limit max_stag, then the algorithm restarts from the initial population based on K replicas of λ_{start}. Note, that in this case the tabu list is not cleared. In [55], max_stag was varied from 100 to 300, while the constants were set to $K = 10, E = 2, M = R = 4$.

Input: Starting point λ_{start}, time limit t, evolution parameters K, E, M, R, max_stag

```
1  P ← initPopulation(λ_start, K)
2  ⟨λ_best, g_best⟩ ← ⟨λ_start, g(λ_start)⟩
3  TabuList ← {λ_start}
4  stag_num ← 0
5  repeat
6      stag_num ← stag_num + 1
7      for each λ in P do
8          if λ not in TabuList then
9              NewFuncValue ← g(λ)
10             addPointTabuList(λ, TabuList)
11             if NewFuncValue < g_best then
12                 ⟨λ_best, g_best⟩ ← ⟨λ, NewFuncValue⟩
13                 stag_num ← 0
14     if stag_num < max_stag then
15         P ← getNextPopulation(P, E, M, R)
16     else
17         stag_num ← 0
18         P ← initPopulation(λ_start, K)
19 until timeExceeded(t)
20 return ⟨λ_best, g_best⟩
```

Algorithm 7: The genetic algorithm

6 Experimental Results

In this section we apply the optimization algorithms described in the previous section to minimization of objective functions introduced in the chapter. As benchmarks we use the problems of finding effective SAT partitionings for several instances of SAT-based cryptanalysis of certain symmetric-key algorithms. In particular, each found BKV corresponds to a guess-and-determine attack on the considered symmetric-key algorithm (see Sect. 3.1). Below we first briefly describe the considered cryptanalysis problems, then say a few words about the implementations of objective functions employed in experiments and finally present the results of experimental evaluation.

6.1 Considered Problems

We consider SAT-based cryptanalysis of several symmetric-key cryptographic algorithms listed in Table 2. In particular, we consider both keystream generators and block ciphers. The keystream generators include: the A5/1 generator (see, e.g., [7]) that is still being used to encrypt traffic in the GSM standard in some countries; the alternating step generator (ASG) characterized by high encryption speed and good statistical properties [28]; the Trivium [10] and Grain_v1 [31] generators that were the finalists of the eSTREAM project aimed at identifying new fast and resistant stream ciphers. The block ciphers include the 2.5-round version of the Advanced Encryption Standard (AES [18]) with 128-bit key, the full (10-round) version of which is one of the most resistant block ciphers used today. Following the notation of [9], "x.5r" means x full rounds and the final round. Another block cipher we study is the 8-round version of the Magma block cipher (also known as GOST 28147-89, see, e.g., [16]). This very weakened version of Magma was studied by SAT-based cryptanalysis in [17, 2]. The full (32-round) version of Magma was used in USSR and Russian cryptographic standards from 1989 to 2019.

Recall that the *known plaintext scenario* [50] in application to a keystream generator means that the attacker has access to a keystream fragment of size m and tries to find the registers' state of size n that produced this fragment. In some cases, the registers' state coincides with the secret key, in other cases it does not, but the secret key can be efficiently computed based on the corresponding state (see, e.g., [48]). Block ciphers construct one block of a ciphertext of a fixed size based on a fixed secret key and one block of a plaintext (as a rule, of the same size as a block of a ciphertext). Known plaintext scenario for block ciphers [50] implies that the attacker has access to known plaintexts and the corresponding ciphertexts, and tries to find the secret key.

It should be noted that the considered objective functions were applied to SAT-based cryptanalysis of several other keystream generators: Bivium (a simplified version of Trivium) and Grain_v0 [62]; Rabbit and Mickey [75]; Salsa20 [74]; Shrinking and Self-shrinking [72]; Bivium and two other simplified versions of

Table 2: Features of studied cryptographic algorithms. Sizes are given in bits.

Keystream generators		
Algorithm	Registers' state	Keystream
A5/1	64	114
AES-96	96	112
Trivium	288	300
Grain_v1	160	200
Block ciphers		
Algorithm	Secret key	Plaintext
AES-128-2.5r	128	384
Magma-8r	256	786

Trivium (Trivium64 and Trivium96) [54]. For brevity, in this study none of them is considered.

All considered cryptanalysis problems were reduced to SAT via the TRANSALG tool [61]. Note that the set X^{in} that is given to some optimization algorithms as the starting point is constructed differently for keystream generators and for block ciphers. More specifically, for keystream generators X^{in} consists of the Boolean variables that encode starting registers' state that has to be found. Recall that a block cipher takes as an input a pair (secret key, plaintext). Let S_1 be a Boolean circuit that specifies a function of the kind $h : \{0,1\}^{n+m} \to \{0,1\}^m$ that corresponds to the considered block cipher. This circuit has $n+m$ inputs, where n is the secret key size in bits, and m – analyzed plaintext size in bits (i.e. m can correspond to several known plaintexts). Note, that in the known plaintext scenario the corresponding plaintext is known. Then, following [61] it can be shown that for this case one can efficiently transit from the circuit S_1 to a circuit S_2 that has n inputs and m outputs. The circuit S_2 specifies some function of the kind (2). If an inversion problem is solved for this function, then the corresponding secret key is found. In this case in the template CNF constructed for S_2 the set X^{in} contains Boolean variables that encode a secret key.

As it was already stated in Sect. 4, in each studied SAT-based cryptanalysis problem X^{in} is a SUPBS for the corresponding SAT instance. This allows us to significantly reduce the search space of the optimization problem. Thus, in the computational experiments the objective functions are minimized over the search spaces of the following sizes: 2^{64} for A5/1; 2^{96} for ASG-96; 2^{288} for Trivium; 2^{160} for Grain_v1; 2^{128} for AES-128-2.5r; 2^{256} for Magma-8r. Therefore, we have 6 hard pseudo-Boolean black-box optimization problems.

6.2 Implementations of Objective Functions

The objective functions (13), (14), and (17) have been implemented in three software tools: PDSAT[1] [62], ALIAS[2] [40], and CryptoEV[3] [55]. They employ different combinations of the objective functions and the algorithms for their minimization. Table 3 shows the optimization algorithms (see Sect. 5) implemented in these tools.

Table 3: Information on software tools implementing the proposed approach.

Tool	Functions	Algorithms
PDSAT	(13), (14), (17)	TSAE, SA
ALIAS	(17)	FCHCVJ, TS, (1+1)-EA
CryptoEV	(13),(14)	(1+1)-EA, GA

The PDSAT tool is implemented in C++, the other two use Python. All these tools use CDCL solvers [46] to solve SAT instances from a SAT partitioning. In the following subsections, the results of all experiments have been obtained using one of the three tools mentioned in Table 3.

6.3 Finding Effective SAT Partitionings

The experiments aimed at minimization of the objective functions (13), (14), and (17) on considered benchmarks were performed in different papers at different times. For these reasons, we are not able to provide the results of each optimization algorithm for minimization of each problem. Thus we will limit the presentation to only published results accompanied by the corresponding citations.

Recall that for each problem all optimization algorithms were given the corresponding Boolean representation of X^{in} as a starting point λ_{start}. For instance, for A5/1 $X^{in} = \{x_1, x_2, \ldots, x_{64}\}$, so $\lambda_{start} = (1, 1, \ldots, 1)$, $|\lambda_{start}| = 64$. Since all studied objective functions are extremely costly, in all trajectory-based optimization algorithms the Hamming distance H was set to 1.

The experiments were conducted on the computing cluster "Academician V.M. Matrosov"[4]. This cluster has nodes of two types – the first one is equipped with 2 × 16-core AMD Opteron 6276 CPUs (32 CPU cores in total) and 64 Gb of RAM, while the second has 2 × 18-core Intel Xeon E5-2695 CPUs and 128 Gb of RAM. Table 4 shows parameters of the conducted experiments. Here "Func." stands for objective function and "Cores" for the number of CPU cores. Note, that in CryptoEv the

[1] https://github.com/Nauchnik/pdsat

[2] https://github.com/Nauchnik/alias

[3] https://github.com/lytr777/CryptoEv

[4] Irkutsk Supercomputer Center of SB RAS, http://hpc.icc.ru

sample size was varied from 10 to 800 [55], while in all other cases it was constant. In the case of ALIAS, three runs of each optimization algorithm were performed, then the best result out of them was chosen. In other cases, there was only one run.

Table 4: Parameters of the performed experiments.

Problem	Func.	Source	Tool	Node type	Cores	Time limit	Sample size (N)
A5/1	(13)	[62]	PDSAT	AMD	160	24 hours	10 000
ASG-96	(14)	[55]	CryptoEv	Intel	180	24 hours	from 10 to 800
	(17)	[73]	PDSAT	Intel	360	12 hours	1 000
Trivium	(14)	[66]	PDSAT	Intel	360	24 hours	1 000
	(17)	[75]	ALIAS	Intel	36	3×24 hours	100
Grain_v1	(17)	[75]	ALIAS	Intel	36	3×24 hours	100
AES-128-2.5	(14)	[66]	PDSAT	Intel	360	24 hours	1 000
Magma-8r	(14)	[66]	PDSAT	Intel	360	24 hours	1 000

Table 5 shows optimization algorithms employed for every pair (problem, objective function), as well as the corresponding BKVs. Fig. 1 shows how the objective function (14) was minimized via (1+1)-EA and GA on ASG-96, while Fig. 2 shows how (17) was minimized via TS, FCHCVJ, and (1+1)-EA on Trivium and Grain_v1. In all these figures, x-axis corresponds to the time in seconds elapsed since the algorithm start, while y-axis corresponds to the objective function's values for λ_{best} in logarithmic scale.

Table 5: Optimization algorithms (and the corresponding found BKVs) applied to each pair (problem, objective function). "-" means that the problem has not been analyzed via the corresponding objective function. For each problem the BKV (and the corresponding algorithm) among all applied objective functions is marked with bold.

Problem	BKV for (13)		BKV for (14)		BKV for (17)	
A5/1	**4.64e+08**	**TSAE** [62]	-	-	-	-
	4.78e+08	SA [62]				
ASG-96	-	-	3.72e+06	GA [55]	**1.06e+06**	**TSAE** [73]
			6.76e+06	(1+1)-EA [55]		
Trivium	-	-	2.04e+41	TSAE [66]	4.46e+43	TS [75]
					2.46e+41	FCHCVJ [75]
					7.15e+40	**(1+1)-EA** [75]
Grain_v1	-	-	-	-	**2.85e+30**	**TS** [75]
					4.07e+30	FCHCVJ [75]
					4.69e+30	(1+1)-EA [75]
AES-128-2.5r	-	-	**1.45e+15**	**TSAE** [66]	-	-
Magma-8r	-	-	**3.55e+22**	**TSAE** [66]	-	-

It is not possible to directly compare all employed optimization algorithms since their different combinations were applied to different problems. However, some

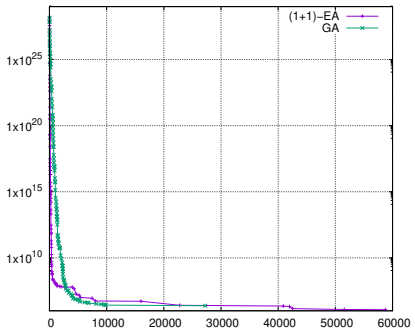
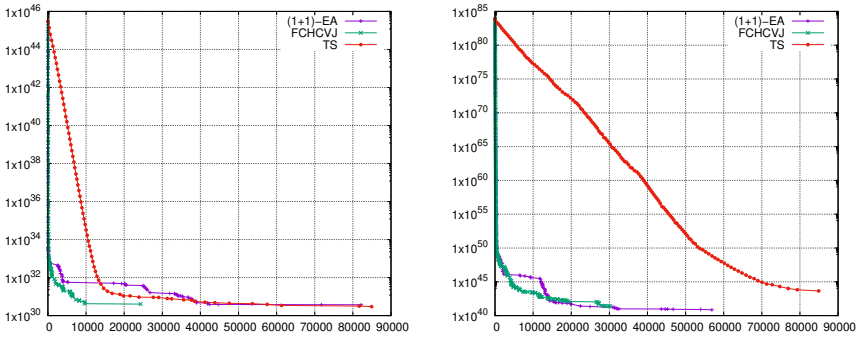


Fig. 1: Minimization of the objective function (14) on ASG-96 by (1+1)-EA and GA.



(a) Grain_v1 (b) Trivium

Fig. 2: Minimization of the objective function (17) on Trivium and Grain_v1 by (1+1)-EA, FCHCVJ, and TS.

conclusions can be made. From the presented results it follows that the algorithms' diversity makes sense for the problems of the considered type. When minimizing the objective function (17) on Grain_v1, the TS, (1+1)-EA, and FCHCVJ algorithms showed comparable results, but TS outperformed its competitors. On Trivium, (1+1)-EA is slightly better than FCHCVJ, while TS showed extremely bad results. The main reason is that TS thoroughly traverses neighborhoods (see Sect. 5), and in the case of Trivium the neighborhoods are quite large. When minimizing the objective function (14) on ASG-96, GA slightly outperformed (1+1)-EA. On A5/1, TSAE showed better results than SA when minimizing the objective function (13). As for AES-128-2.5r and Magma-8r, only TSAE was employed for each of them.

6.4 Solving Hard SAT Instances via Found Partitionings

To make sure that the constructed runtime estimations are accurate, we solved several SAT-based cryptanalysis instances via the constructed SAT partitionings. Recall that each SAT partitioning is constructed using a decomposition set that is defined by some λ (see Sect. 3). We used λ_{best} found for A5/1 and ASG-96 (see the best results in Table 5) to implement SAT-based guess-and-determine attacks on these keystream generators. This choice was motivated by the fact that they correspond to the smallest runtime estimations among all presented in Table 5.

For the first time SAT-based cryptanalysis of A5/1 was considered in [65]. In that paper, the decomposition set consisting of 31 variables (out of 64) was constructed manually based on the analysis of A5/1 algorithmic features. Using this partitioning, 10 SAT-based cryptanalysis instances for A5/1 were solved [62] in the volunteer computing project SAT@home [56]. Also, in SAT@home the same 10 instances were solved by the decomposition set found via TSAE when minimizing the objective function (13) [62]. Note, that this decomposition set consists of 32 variables. In both experiments the average runtime on simplified instances from the corresponding SAT partitionings was comparable.

As for ASG-96, 20 SAT-based cryptanalysis instances were solved on the computing cluster "Academician V.M. Matrosov" via the decomposition set found by the TSAE optimization algorithm when minimizing (17) [73]. The decomposition set consisted of 30 variables (out of 96). Is should be noted, that the corresponding SAT partitioning significantly outperformed by average runtime the one constructed manually [73].

According to the best obtained estimations, Trivium, Grain_v1, AES-128-2.5r, and Magma-8r are way too hard for processing the corresponding SAT partitionings in reasonable time. Following [62, 61], we studied weakened SAT-based cryptanalysis problems for Trivium and Grain_v1 by assigning correct values to a portion of variables from X^{in} [75]. In particular, we assigned values to k (out of n) last variables from X^{in}. In such a weakened variant, the first $n-k$ variables from X^{in} are unknown, thus, λ_{best} is picked from the set of all possible subsets of $\{x_1, \ldots, x_{n-k}\}$. The following weakened variants were considered: $k = 96$ for Grain_v1 and $k = 134$ for Trivium, so λ_{best} was picked from subsets of $\{x_1, \ldots, x_{64}\}$ and $\{x_1, \ldots, x_{154}\}$, respectively. (1+1)-EA was run on both weakened problems for 1 hour on one cluster node. As a result, for Grain_v1 (Trivium) λ_{best} consisting of 22 (26) variables was found. By processing the corresponding SAT partitionings, for both Grain_v1 and Trivium three randomly generated weakened SAT-based cryptanalysis problems were successfully solved on one cluster node.

It turned out, that for each solved problem the runtime estimation is accurate since it is quite close to the corresponding real solving time.

7 Conclusion

In this chapter we discussed how the black-box optimization methods can be applied to finding good partitionings for hard instances of the Boolean satisfiability problem. We narrow this area to the cases when a special partitioning method can be applied, since it allows one to formulate the problem of finding effective partitionings as a black-box optimization problem. The presented approach works quite well with cryptanalysis problems in SAT form, and sometimes leads to state-of-the-art results. In contrast to the majority of cryptographic attacks that require in-depth analysis of the underlying cipher and a lot of manual tinkering with peculiarities of a particular construction, the approach presented in the chapter is completely automatic. We give a problem formulation to a black-box optimization algorithm, launch it and wait while it finds a minimum. While it is by no means a silver bullet, and the approach has some pitfalls, the results of experiments appear to be quite promising. In combination with the relevant research in neighboring areas, the presented results make us hope that the potential of SAT in the context of solving hard combinatorial problems is far from being exhausted.

Acknowledgements The research was funded by Russian Science Foundation (project No. 16-11-10046). Stepan Kochemazov is additionally supported by the Council for Grants of the President of the Russian Federation (stipend SP-2017.2019.5).

References

1. Audet, C., Hare, W.: Derivative-Free and Blackbox Optimization. Springer Series in Operations Research and Financial Engineering. Springer International Publishing, Berlin (2017). DOI 10.1007/978-3-319-68913-5
2. Babenko, L.K., Maro, E.A., Anikeev, M.V.: Application of algebraic cryptanalysis to MAGMA and PRESENT block encryption standards. In: Proceedings of IEEE 11th International Conference on Application of Information and Communication Technologies (AICT), pp. 1–7 (2017). DOI 10.1109/ICAICT.2017.8686954
3. Balyo, T., Sinz, C.: Parallel satisfiability. In: Y. Hamadi, L. Sais (eds.) Handbook of Parallel Constraint Reasoning, pp. 3–29. Springer (2018). DOI 10.1007/978-3-319-63516-3_1
4. Bard, G.V.: Algebraic Cryptanalysis, 1st edn. Springer Publishing Company, Incorporated (2009)
5. Bessiere, C., Katsirelos, G., Narodytska, N., Walsh, T.: Circuit complexity and decompositions of global constraints. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence - IJCAI'09, pp. 412–418 (2009)
6. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, *Frontiers in Artificial Intelligence and Applications*, vol. 185. IOS Press (2009)
7. Biryukov, A., Shamir, A., Wagner, D.A.: Real time cryptanalysis of A5/1 on a PC. In: B. Schneier (ed.) Fast Software Encryption, 7th International Workshop, FSE 2000, *Lecture Notes in Computer Science*, vol. 1978, pp. 1–18. Springer (2000). DOI 10.1007/3-540-44706-7_1
8. Boros, E., Hammer, P.L.: Pseudo-boolean optimization. Discrete Appl. Math. **123**(1-3), 155–225 (2002)

9. Bouillaguet, C., Derbez, P., Fouque, P.: Automatic search of attacks on round-reduced AES and applications. In: P. Rogaway (ed.) Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, *Lecture Notes in Computer Science*, vol. 6841, pp. 169–187. Springer (2011). DOI 10.1007/978-3-642-22792-9_10
10. Cannière, C.D., Preneel, B.: Trivium. In: M.J.B. Robshaw, O. Billet (eds.) New Stream Cipher Designs - The eSTREAM Finalists, *Lecture Notes in Computer Science*, vol. 4986, pp. 244–266. Springer (2008). DOI 10.1007/978-3-540-68351-3_18. URL https://doi.org/10.1007/978-3-540-68351-3_18
11. Carter, K., Foltzer, A., Hendrix, J., Huffman, B., Tomb, A.: SAW: the software analysis workbench. In: J. Boleng, S.T. Taft (eds.) Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology, HILT, pp. 15–18. ACM (2013). DOI 10.1145/2527269.2527277
12. Chang, C.L., Lee, R.C.T.: Symbolic Logic and Mechanical Theorem Proving, 1st edn. Academic Press, Inc., USA (1997)
13. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: K. Jensen, A. Podelski (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, *Lecture Notes in Computer Science*, vol. 2988, pp. 168–176. Springer (2004). DOI 10.1007/978-3-540-24730-2_15
14. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, pp. 151–158 (1971)
15. Cook, S.A., Mitchell, D.G.: Finding hard instances of the satisfiability problem: A survey. In: Satisfiability Problem: Theory and Applications, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 35, pp. 1–18 (1996)
16. Courtois, N.T.: Algebraic complexity reduction and cryptanalysis of GOST. IACR Cryptol. ePrint Arch. **2011**, 626 (2011). URL http://eprint.iacr.org/2011/626
17. Courtois, N.T., Gawinecki, J.A., Song, G.: Contradiction immunity and guess-then-determine attacks on GOST. Tatra Mt. Math. Publ. **53**(1), 2–13 (2012)
18. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Information Security and Cryptography. Springer (2002). DOI 10.1007/978-3-662-04722-4
19. Dowling, W.F., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional horn formulae. J. Log. Program. **1**(3), 267–284 (1984)
20. Eén, N., Sörensson, N.: An extensible SAT-solver. In: E. Giunchiglia, A. Tacchella (eds.) Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Selected Revised Papers, *Lecture Notes in Computer Science*, vol. 2919, pp. 502–518. Springer (2003). DOI 10.1007/978-3-540-24605-3_37
21. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. **89**(4), 543–560 (2003)
22. Eibach, T., Pilz, E., Völkel, G.: Attacking bivium using SAT solvers. In: H.K. Büning, X. Zhao (eds.) Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, *Lecture Notes in Computer Science*, vol. 4996, pp. 63–76. Springer (2008). DOI 10.1007/978-3-540-79719-7_7
23. Feller, W.: An introduction to probability theory and its applications, Volume II. John Wiley & Sons Inc., New York, NY, USA (1971)
24. Franco, J., Martin, J.: A history of satisfiability. In: A. Biere, M. Heule, H. van Maaren, T. Walsh (eds.) Handbook of Satisfiability, *Frontiers in Artificial Intelligence and Applications*, vol. 185, pp. 3–74. IOS Press (2009). DOI 10.3233/978-1-58603-929-5-3. URL https://doi.org/10.3233/978-1-58603-929-5-3
25. Garey, M.R., Johnson, D.S.: Computers and intractability, vol. 174. Freeman New York (1979)
26. Glover, F.: Future paths for integer programming and links to artificial intelligence. Computers & OR **13**(5), 533–549 (1986)
27. Gomes, C.P., Sabharwal, A.: Exploiting runtime variation in complete solvers. In: A. Biere, M. Heule, H. van Maaren, T. Walsh (eds.) Handbook of Satisfiability, *Frontiers in Artificial Intelligence and Applications*, vol. 185, pp. 271–288. IOS Press (2009). DOI 10.3233/978-1-58603-929-5-271. URL https://doi.org/10.3233/978-1-58603-929-5-271

28. Günther, C.G.: Alternating step generators controlled by de Bruijn sequences. In: D. Chaum, W.L. Price (eds.) Advances in Cryptology - EUROCRYPT '87, Workshop on the Theory and Application of of Cryptographic Techniques, *Lecture Notes in Computer Science*, vol. 304, pp. 5–14. Springer (1987). DOI 10.1007/3-540-39118-5_2
29. Hamadi, Y., Jabbour, S., Sais, L.: Manysat: a parallel SAT solver. J. Satisf. Boolean Model. Comput. **6**(4), 245–262 (2009)
30. Hamming, R.W.: Error detecting and error correcting codes. The Bell System Technical Journal **29**(2), 147–160 (1950). DOI 10.1002/j.1538-7305.1950.tb00463.x
31. Hell, M., Johansson, T., Maximov, A., Meier, W.: The grain family of stream ciphers. In: M.J.B. Robshaw, O. Billet (eds.) New Stream Cipher Designs - The eSTREAM Finalists, *Lecture Notes in Computer Science*, vol. 4986, pp. 179–190. Springer (2008). DOI 10.1007/978-3-540-68351-3_14. URL https://doi.org/10.1007/978-3-540-68351-3_14
32. Heule, M., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: K. Eder, J. Lourenço, O. Shehory (eds.) Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, *Lecture Notes in Computer Science*, vol. 7261, pp. 50–65. Springer (2011). DOI 10.1007/978-3-642-34188-5_8
33. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In: N. Creignou, D. Le Berre (eds.) Theory and Applications of Satisfiability Testing – SAT 2016, *Lecture Notes in Computer Science*, vol. 9710, pp. 228–245 (2016)
34. Hyvärinen, A.E.J.: Grid Based Propositional Satisfiability Solving. Ph.D. thesis, Aalto University (2011)
35. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning SAT instances for distributed solving. In: C.G. Fermüller, A. Voronkov (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, LPAR-17, pp. 372–386 (2010). DOI 10.1007/978-3-642-16242-8_27
36. Janicic, P.: URSA: a system for uniform reduction to SAT. Log. Meth. Comput. Sci. **8**(3), 1–39 (2012)
37. Järvisalo, M., Biere, A., Heule, M.: Simulating circuit-level simplifications on CNF. J. Autom. Reasoning **49**(4), 583–619 (2012)
38. Järvisalo, M., Junttila, T.: Limitations of restricted branching in clause learning. Constraints **14**(3), 325–356 (2009)
39. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**(4598), 671–680 (1983)
40. Kochemazov, S., Zaikin, O.: ALIAS: A modular tool for finding backdoors for SAT. In: O. Beyersdorff, C.M. Wintersteiger (eds.) Theory and Applications of Satisfiability Testing - 21st International Conference, SAT 2018, *Lecture Notes in Computer Science*, vol. 10929, pp. 419–427. Springer (2018). DOI 10.1007/978-3-319-94144-8_25
41. Kolda, T.G., Lewis, R.M., Torczon, V.: Optimization by direct search: New perspectives on some classical and modern methods. SIAM review **45**(3), 385–482 (2003)
42. Kroening, D.: Software verification. In: A. Biere, M. Heule, H. van Maaren, T. Walsh (eds.) Handbook of Satisfiability, *Frontiers in Artificial Intelligence and Applications*, vol. 185, pp. 505–532. IOS Press (2009). DOI 10.3233/978-1-58603-929-5-505. URL https://doi.org/10.3233/978-1-58603-929-5-505
43. Lafitte, F.: Cryptosat: a tool for SAT-based cryptanalysis. IET Inf. Secur. **12**(6), 463–474 (2018). DOI 10.1049/iet-ifs.2017.0176. URL https://doi.org/10.1049/iet-ifs.2017.0176
44. Levin, L.: Universal sequential search problems. Problems Inform. Transmission **9**, 265–266 (1973)
45. Luke, S.: Essentials of Metaheuristics, second edn. Lulu (2013). Available for free at http://cs.gmu.edu/~sean/book/metaheuristics/
46. Marques-Silva, J.P., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: A. Biere, M. Heule, H. van Maaren, T. Walsh (eds.) Handbook of Satisfiability, *Frontiers in Artificial Intelligence and Applications*, vol. 185, pp. 131–153. IOS Press (2009). DOI 10.3233/978-1-58603-929-5-131. URL https://doi.org/10.3233/978-1-58603-929-5-131

47. Marques-Silva, J.P., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: R.A. Rutenbar, R.H.J.M. Otten (eds.) Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, pp. 220–227. IEEE Computer Society / ACM (1996). DOI 10.1109/ICCAD.1996.569607
48. Maximov, A., Biryukov, A.: Two trivial attacks on trivium. In: C.M. Adams, A. Miri, M.J. Wiener (eds.) Selected Areas in Cryptography, 14th International Workshop, SAC 2007, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 4876, pp. 36–55. Springer (2007). DOI 10.1007/978-3-540-77360-3_3
49. Mcdonald, C., Charnes, C., Pieprzyk, J.: Attacking Bivium with MiniSat. Tech. Rep. 2007/040, ECRYPT Stream Cipher Project (2007)
50. Menezes, A.J., Vanstone, S.A., Oorschot, P.C.V.: Handbook of Applied Cryptography, 1st edn. CRC Press, Inc., Boca Raton, FL, USA (1996)
51. Metropolis, N., Ulam, S.: The Monte Carlo Method. J. Amer. statistical assoc. **44**(247), 335–341 (1949)
52. Mühlenbein, H.: How genetic algorithms really work: Mutation and hillclimbing. In: R. Männer, B. Manderick (eds.) Parallel Problem Solving from Nature 2, PPSN-II, pp. 15–26. Elsevier (1992)
53. Otpuschennikov, I.V., Semenov, A.A., Gribanova, I., Zaikin, O., Kochemazov, S.: Encoding cryptographic functions to SAT using TRANSALG system. In: G.A. Kaminka, M. Fox, P. Bouquet, E. Hüllermeier, V. Dignum, F. Dignum, F. van Harmelen (eds.) ECAI 2016 - 22nd European Conference on Artificial Intelligence, *Frontiers in Artificial Intelligence and Applications*, vol. 285, pp. 1594–1595. IOS Press (2016). DOI 10.3233/978-1-61499-672-9-1594
54. Pavlenko, A., Buzdalov, M., Ulyantsev, V.: Fitness comparison by statistical testing in construction of SAT-based guess-and-determine cryptographic attacks. In: A. Auger, T. Stützle (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, pp. 312–320 (2019). DOI 10.1145/3321707.3321847
55. Pavlenko, A., Semenov, A.A., Ulyantsev, V.: Evolutionary computation techniques for constructing SAT-based attacks in algebraic cryptanalysis. In: P. Kaufmann, P.A. Castillo (eds.) Applications of Evolutionary Computation - 22nd International Conference, EvoApplications 2019, *Lecture Notes in Computer Science*, vol. 11454, pp. 237–253. Springer (2019). DOI 10.1007/978-3-030-16692-2_16
56. Posypkin, M., Semenov, A.A., Zaikin, O.: Using BOINC desktop grid to solve large scale SAT problems. Computer Science (AGH) **13**(1), 25–34 (2012)
57. Rios, L., Sahinidis, N.: Derivative-free optimization: A review of algorithms and comparison of software implementations. J. Global Optim. **56**, 1247–1293 (2013). DOI 10.1007/s10898-012-9951-y
58. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965). DOI 10.1145/321250.321253
59. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice Hall (2009)
60. Semenov, A.: Decomposition representations of logical equations in problems of inversion of discrete functions. J. Comput. Syst. Sci. Int. **48**, 718–731 (2009)
61. Semenov, A., Otpuschennikov, I., Gribanova, I., Zaikin, O., Kochemazov, S.: Translation of algorithmic descriptions of discrete functions to SAT with applications to cryptanalysis problems. Log. Meth. Comput. Sci. **16** (2020)
62. Semenov, A., Zaikin, O.: 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)
63. Semenov, A.A., Zaikin, O.: Using Monte Carlo method for searching partitionings of hard variants of Boolean satisfiability problem. In: V. Malyshkin (ed.) Parallel Computing Technologies - 13th International Conference, PaCT 2015, *Lecture Notes in Computer Science*, vol. 9251, pp. 222–230. Springer (2015). DOI 10.1007/978-3-319-21909-7_21
64. Semenov, A.A., Zaikin, O.: On the accuracy of statistical estimations of SAT partitionings effectiveness in application to discrete function inversion problems. In: A.V. Kononov, I.A.

Bykadorov, O.V. Khamisov, I.A. Davydov, P.A. Kononova (eds.) Supplementary Proceedings of the 9th International Conference on Discrete Optimization and Operations Research and Scientific School (DOOR 2016), *CEUR Workshop Proceedings*, vol. 1623, pp. 261–275. CEUR-WS.org (2016)

65. Semenov, A.A., Zaikin, O., Bespalov, D., Posypkin, M.: Parallel logical cryptanalysis of the generator A5/1 in bnb-grid system. In: V. Malyshkin (ed.) Parallel Computing Technologies - 11th International Conference, PaCT 2011, *Lecture Notes in Computer Science*, vol. 6873, pp. 473–483. Springer (2011). DOI 10.1007/978-3-642-23178-0_43
66. Semenov, A.A., Zaikin, O., Otpuschennikov, I.V., Kochemazov, S., Ignatiev, A.: On cryptographic attacks using backdoors for SAT. In: S.A. McIlraith, K.Q. Weinberger (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), pp. 6641–6648. AAAI Press (2018)
67. Soos, M.: Grain of Salt - an automated way to test stream ciphers through SAT solvers. In: Tools'10: Proceedings of the Workshop on Tools for Cryptanalysis, pp. 131–144 (2010)
68. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: O. Kullmann (ed.) Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, *Lecture Notes in Computer Science*, vol. 5584, pp. 244–257. Springer (2009). DOI 10.1007/978-3-642-02777-2_24
69. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: A.O. Slisenko (ed.) Studies in mathematics and mathematical logic, Part II, pp. 115—-125. Steklov Mathematical Institute (1968)
70. Wegener, I.: The Complexity of Boolean Functions. John Wiley & Sons, Inc., USA (1987)
71. Williams, R., Gomes, C.P., Selman, B.: Backdoors to typical case complexity. In: G. Gottlob, T. Walsh (eds.) Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, IJCAI-03, pp. 1173–1178. Morgan Kaufmann (2003)
72. Zaikin, O.: SAT-based cryptanalysis: From parallel computing to volunteer computing. In: V.V. Voevodin, S. Sobolev (eds.) Supercomputing - 5th Russian Supercomputing Days, RuSCDays 2019, *Communications in Computer and Information Science*, vol. 1129, pp. 701–712. Springer (2019). DOI 10.1007/978-3-030-36592-9_57
73. Zaikin, O., Kochemazov, S.: An improved SAT-based guess-and-determine attack on the alternating step generator. In: P.Q. Nguyen, J. Zhou (eds.) Information Security - 20th International Conference, ISC 2017, *Lecture Notes in Computer Science*, vol. 10599, pp. 21–38. Springer (2017). DOI 10.1007/978-3-319-69659-1_2
74. Zaikin, O., Kochemazov, S.: Pseudo-boolean black-box optimization methods in the context of divide-and-conquer approach to solving hard SAT instances. In: OPTIMA 2018 (Supplementary Volume), pp. 76–87. DEStech Publications, Inc. (2018)
75. Zaikin, O., Kochemazov, S.: On black-box optimization in divide-and-conquer SAT solving. Optimization Methods and Software pp. 1–25 (2019). DOI 10.1080/10556788.2019.1685993. URL https://doi.org/10.1080/10556788.2019.1685993
76. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. J. Symb. Comput. **21**(4), 543–560 (1996). DOI 10.1006/jsco.1996.0030