Pseudo-Boolean Black-Box Optimization Methods in the Context of Divide-and-Conquer Approach to Solving Hard SAT Instances

Oleg ZAIKIN$^{1,2,*}$ and Stepan KOCHEMAZOV$^1$

$^1$ Matrosov Institute for System Dynamics and Control Theory SB RAS, 664033, Irkutsk, Russia  
$^2$ ITMO University, 197101, St Petersburg, Russia  
$^*$ Corresponding author

Keywords: Black-box optimization, Pseudo-Boolean optimization, SAT, Divide-and-conquer, Cryptanalysis, Stream cipher.

Abstract. Solving hard instances of the Boolean satisfiability problem (SAT) in practice is an interestingly nontrivial area. In particular, the heuristic nature of state-of-the-art SAT solvers makes it impossible to know in advance how long it will take to solve any particular SAT instance. One way of coping with this disadvantage is the divide-and-conquer approach when an original SAT instance is decomposed into several simpler subproblems. However, the way it is decomposed plays a crucial role in the resulting effectiveness of solving. In the present study, we reduce the problem of “dividing” a hard SAT instance into many simpler subproblems to a pseudo-Boolean black-box optimization problem. Then we use relevant implementations of corresponding methods to analyze hard SAT instances encoding the cryptanalysis of state-of-the-art stream ciphers proposed during the recent e-STREAM competition.

Introduction

Boolean satisfiability problem (SAT) [4] is historically the first NP-complete problem. It means that there are numerous practical problems that can be reduced to SAT. Moreover, the progress in SAT-solving algorithms in recent two decades made it possible to reach a stage when state-of-the-art SAT solvers can cope with various industrial and combinatorial problems in reasonable time. One of the areas where SAT solvers allow to obtain quite interesting results is cryptanalysis. The corresponding approach is called SAT-based cryptanalysis. The reason why SAT is useful for cryptographic problems is that they essentially deal with low-level data transformations which are hard to invert. Thus it is natural to reduce them to SAT and attempt to solve them in that form.

In this study, we analyze one specific facet of SAT-based cryptanalysis. In particular, when solving hard SAT instances, e.g. the ones encoding the cryptanalysis of some ciphers, one of the commonly used techniques is the so-called divide-and-conquer approach. Informally, it consists in splitting an original problem into a (large) family of simpler subproblems, preferably in such a way that their resulting solving time is less than that for an original problem. An interesting fact is that the latter is not obligatory, because the divide-and-conquer approach enables the use of distributed computing resources compared to launching SAT solver on a target instance on a single processor (core) [45]. The non-trivial issue is that the way a problem is split into a family of subproblems has a direct impact on the time it takes to solve it. In several papers, including
it was shown that it is possible to represent the problem of how to split a hard SAT instance into simpler tasks as a problem of pseudo-Boolean black-box optimization.

The goal of the present paper is to answer the question whether it is necessary to design specific algorithms for solving the outlined black-box problem or the existing widely available tools are good enough to cope even with splitting very hard SAT instances which encode the cryptanalysis of state-of-the-art stream ciphers. In the role of the latter, we analyzed several finalists of e-STREAM project, a cryptographic competition, aimed at finding new stream ciphers that could replace the existing ones in the nearest future.

Let us give a brief outline of the paper. In the next section, we introduce a number of notations, formulate and discuss the objective pseudo-Boolean black-box function that will later be optimized using different algorithms. In the third section, we describe the optimization algorithms that we employ to optimize the objective function and their features. The fourth section contains detailed information on the considered problems and the cryptographic specifics. In the fifth section, we present the results of computational experiments. In the last two sections, we discuss the results and draw conclusions.

Black-box Optimization and Hard SAT Instances

Boolean satisfiability problem (SAT) consists in the following: for an arbitrary Boolean formula to decide if there is an assignment of its variables that makes this formula True, meaning that a formula is satisfiable, or to prove that there are no such assignments and thus a formula is unsatisfiable.

There are many ways to divide a hard SAT instance into a family of simpler subproblems. Some of them were described, for example, in [23]. In the present paper we use the so-called plain partitioning approach (in terms of [23]), that was used in [35, 12, 43, 25], etc. Let us briefly describe it below.

Assume that $C$ is a Boolean formula in a conjunctive normal form (CNF) over a set of $n$ Boolean variables $X$. We choose a subset $S$ of variables from $X$. Since it contains Boolean variables, it means that there are exactly $2^{|S|}$ possible ways to assign values $\in \{True, False\}$ to variables from $S$ in $C$. When we assign values $\alpha = (\alpha_1, \ldots, \alpha_k)$ to variables from a set $S$, $|S|$, let us denote a simplified SAT instance as $C[\alpha/S]$. Let us refer to a set of subproblems produced by instantiating variables from $S$ in $C$ as to a decomposition of an original SAT instance $C$ and denote it as $D_S[C] = \{C[\alpha/S], \alpha \in \{0, 1\}^{|S|}\}$.

Naturally, it is possible to choose a set $S$ in such a way (in the simplest case by varying its size $|S|$), that the resulting subproblems can be made as simple for a SAT solver as necessary. The nontrivial moment consists in that different sets of the same size can produce vastly different subproblems complexity-wise. I.e. it is entirely possible that for some $S_1, S_2 \subseteq X$, $S_1 \neq S_2$, $|S_1| = |S_2|$ the total time to solve all the subproblems from the decomposition $D_{S_1}[C]$ using some specific SAT solver can be significantly less than that for $D_{S_2}[C]$. That is why it is important to pick a good set $S$ to decompose a problem. One possible way of doing this is via blackbox optimization. Ideally, the value of an objective function on a set $S$ would for a particular solver $G$ represent the sum of runtimes of the solver on all subproblems from $D_{S}$. However, this approach is impractical since it implies solving the same problem multiple times. And what if the problem is too complex to be solved? Realistically, the value of an objective function should be computed if not effectively, then at least relatively fast.
In the remainder of the paper the objective function value is computed using the Monte-Carlo method [30] as follows. Assume there is a SAT instance $C$, a set $S$ we use to construct a decomposition $D_S[C]$ and a solver $G$ we employ to solve simplified subproblems. Our goal is to estimate the time it would take to solve all subproblems from $D_S[C]$ with the help of $G$. For this purpose, in accordance with the Monte Carlo method first construct a random sample $RS = \{C[\beta_1/S], \ldots, C[\beta_N/S]\}$ of size $N$ by choosing randomly $\{\beta_1, \ldots, \beta_N\}$ from $\{0, 1\}^{|S|}$. Then launch the solver $G$ on each subproblem from $RS$. Assume that $T_G(C[\beta_i/S])$ is the runtime of $G$ on $C[\beta_i/S]$. Then, the objective function is computed as follows [25]:

$$Runtime_{\text{Estimation}}_{C,B,N}(S) = 2^{|S|} \times \frac{1}{N} \times \sum_{i=1}^{N} T_G(C[\beta_i/S]). \quad (1)$$

Basically, (1) implements in the straightforward manner the Monte Carlo method to compute the runtime estimation for solving a particular hard SAT instance by decomposing it via a particular set $S$.

Note, that (1) is in fact a black-box objective function. It is also a pseudo-Boolean objective function, because its value is a real number (an estimation in seconds), while $S$ is a subset of a set of Boolean variables $X, |X| = n$, so each possible $S$ corresponds to some Boolean vector of length $n$. Finally, the objective function can be represented as follows:

$$Runtime_{\text{Estimation}}_{C,B,N}(S) : B^n \rightarrow \mathbb{R}. \quad (2)$$

In the following section, we discuss what requirements should the algorithms for minimization of the objective function (1) satisfy.

**Optimization Algorithms Involved**

Assume that a CNF over a set of Boolean variables $X$ is given. Then in the general case, a search space has $2^{|X|}$ points, each corresponding to some subset $S$ of set $X$. Thus, for any of them, it is possible to compute the value of the objective function (1). Hence, a black-box optimization algorithm can be used to traverse a search space. It is important to note, that a search space can be significantly reduced, if, for example, some variables in CNF are dependent on others. When a SAT instance encodes some circuit with explicit inputs and outputs, then instantiating the variables corresponding to inputs usually leads to the derivation of all the other variables in a formula. It makes sense to construct a search space based only on such independent variables. In all problems considered in Sections 4 and 5 such knowledge was available.

It is clear that not just any optimization algorithm can be applied to minimization of function (1). To be used in practice, it should satisfy several criteria:

1. It should be able to operate with a black-box objective function;
2. It should be able to work with Boolean (or pseudo-Boolean) variables;
3. There should be a possibility to launch it on a computing cluster equipped with standard compilers and software packages.

The first two criteria are quite natural for the objective function (1). The third criterion was chosen because computing the value of (1) at any point is usually quite expensive. For example, depending on the problem and the size of the random sample, it can take up to several minutes to compute (1) for one subset $S$ on a computer equipped with multi-core CPU. Thus, to deal with the
hard optimization problems that will be further described in Section 4, it is necessary to employ a computing cluster. Since usually on a cluster only standard user rights are allowed, it is quite hard and sometimes impossible to install any nonstandard additional software on it. This restriction, in turn, implies that optimization algorithms must be implemented on widely used programming languages (C++, Python, Java, Fortran, etc.) maintained by modern computing clusters.

The vast majority of optimization algorithms cannot operate with black-box objective functions, so they do not satisfy the first criterion. A number of black-box optimization algorithms don’t satisfy the second one, because they are designed to deal only with continuous variables. For instance, it holds true for Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [16], Hooke-Jeeves method [19], pseudo-gradient approach [13] and a variety of coordinate descent techniques. Finally, almost all black-box discrete optimization algorithms fail the third criterion. For instance, Multilevel Coordinate Search (MSC) [22] has an only Matlab implementation, so it cannot be launched on a standard computing cluster. In the case of Radial Basis Function algorithm for black-box optimization (RBFOpt [11]) there exists its Python implementation, but RBFOpt must be installed on a cluster. The latter is impossible because it cannot be done using ordinary cluster’s users rights.

The only general-purpose optimization algorithm we have found so far which satisfies all three criteria is the Sequential Model-based Algorithm Configuration (SMAC) [21]. Also in paper [25] there was described the ALIAS tool which implements a variant of Greedy Best First Search (GBFS) algorithm designed specifically for minimization of function [1]. Of course, it is also possible to implement specifically tailored versions of evolutionary or genetic algorithms [41], as well as variable neighborhood search [17] or tabu search [15], as well as a number of other heuristics. However, the main goal was to take existing implementations and use them in the study. In the following subsections, we briefly describe both SMAC and the GBFS algorithms we employed in our experiments.

Sequential Model-Based Optimization for General Algorithm Configuration

SMAC (sequential model-based algorithm configuration) [21] is a tool that can be used both for black-box optimization with mixed (continuous and discrete) variables, and for solving algorithm configuration problem (to find a set of parameter values which yield the best performance of an algorithm in given conditions). The predictive models SMAC is based on can also capture and exploit important information about the model domain, such as which input variables are most important. SMAC is an implementation of the Sequential Model-based optimization (SMBO) framework [24]. According to SMBO, a regression model is constructed, that predicts values of an objective function and then this model is used for optimization. SMBO has been used for optimizing costly black-box functions. SMAC is based on the random forest machine learning algorithm [6]. In fact, a random forest is a collection of either decision trees or regression trees. In SMAC random forest contains regression trees that have real values (values of an objective function) at their leaves. Every time a new value of an objective function is calculated, a random forest is reconstructed. In fact, random forest recommends in which points an objective function should be calculated. SMAC is implemented in Python and Java.
Greedy Best First Search in ALIAS

As it was mentioned in Section 2, each possible subset \( S \) of a set of Boolean variables \( X \), \( |X| = n \) of a given CNF, corresponds to a Boolean vector of length \( n \). That is why any black-box local search algorithm that operates with Hamming distances can be employed to minimize the considered objective function. As such algorithm, the Greedy Best First Search (GBFS) algorithm [34] was chosen. The GBFS algorithm described below was implemented as a module of the ALIAS tool [25].

GBFS starts from a starting point, in the role of which it is possible to use the whole set of Boolean variables \( X \), to obtain a baseline runtime estimation (for such a point it can always be computed effectively). Then the algorithm checks all points from the neighborhood of the starting point (a set of points at Hamming distance of 1). If it finds a better point, then it starts checking its neighborhood. If all points from a neighborhood are worse than the current best-known value, then it means that a local minimum is reached. Since the computation of the objective function in an arbitrary point is costly, the points for which the function value was computed are stored in special lists in order to not compute function at any point twice.

When a local minimum is reached, in [25] the following simple jump strategy is used: the point with the current best-known value is partially randomly permuted and the obtained point is used as a new starting point. The algorithm stops either if the time limit is exceeded, or if the limit on jumps is reached. It turned out, that this strategy usually does not lead to updating the best-known value. That is why in the present study another jump strategy is used. According to it, \( 2k \) Boolean variables are added to a record point. Among them, \( k \) variables are the ones, which occurred in the least number of points of the search space, in which the value [1] was computed. In the role of another \( k \) variables the ones which participate in the largest number of record points are chosen. In all experiments described in Section 5 \( k \) was equal to 4. It turned out, that this strategy shows much better results than the one used in [25].

Considered Problems

In the role of hard SAT instances, we considered several instances of cryptanalysis of cryptographic keystream generators. A cryptographic keystream generator is a special mathematical design that is used to construct a pseudorandom bit sequence that is later used to cipher known data via bitwise XOR [29]. The generated bit sequence is usually referred to as keystream. Cryptographic keystream generators are usually designed in such a way that knowing their inner workings gives attacker little to no leverage without the knowledge of (at least a part of) secret key or inordinate amounts of keystream. We considered the problem of cryptanalysis of keystream generators in the following form: given a known fragment of keystream to recover the values of a generator registers cells that were used to generate this keystream. The problems of cryptanalysis of keystream generators in SAT form can be viewed as almost a perfect example of hard SAT instances – they are quite compact but at the same time very hard.

We analyzed five keystream generators — finalists in the eSTREAM project [8]. This project was organized by European cryptological community and was aimed at identifying new stream ciphers that could in the nearest future replace existing ones. Thus the e-STREAM’s finalists can be considered as state of the art in the area of stream ciphers. The generators analyzed below are called Trivium [7], Grain [18], Mickey [1], Rabbit [5] and Salsa20 [3]. We will refrain from providing detailed descriptions for these generators because they are not entirely relevant to the
Table 1. Characteristics of SAT encodings of the considered cryptanalysis problems.

<table>
<thead>
<tr>
<th>Generator</th>
<th>Variables</th>
<th>Clauses</th>
<th>Size (Mb)</th>
<th>Registers size (bits)</th>
<th>Keystream size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trivium</td>
<td>1 887</td>
<td>22 776</td>
<td>0.5</td>
<td>288</td>
<td>300</td>
</tr>
<tr>
<td>Grain</td>
<td>1 945</td>
<td>37 542</td>
<td>1.1</td>
<td>160</td>
<td>160</td>
</tr>
<tr>
<td>Mickey</td>
<td>72 078</td>
<td>586 080</td>
<td>15.7</td>
<td>200</td>
<td>250</td>
</tr>
<tr>
<td>Rabbit</td>
<td>98 449</td>
<td>879 713</td>
<td>23.7</td>
<td>513</td>
<td>512</td>
</tr>
<tr>
<td>Salsa20</td>
<td>26 976</td>
<td>989 912</td>
<td>44.8</td>
<td>512</td>
<td>512</td>
</tr>
</tbody>
</table>

In order to construct the SAT encodings we employed the TRANSALG software system \[32\]. It uses a C-like language, thus, thanks to the fact that eSTREAM candidates provided C implementations of each generator, it was relatively simple to transform the latter into programs for TRANSALG and test their correctness. The data on obtained SAT encodings is presented in Table 1.

Computational Experiments

For each of 5 considered keystream generators (see Section 4) 1 cryptanalysis instance was considered. The obtained instances were encoded to SAT in a way described in Section 4 resulting in 5 hard SAT instances, which in turn correspond to 5 hard black-box pseudo-Boolean optimization problems.

As a starting point for every problem, a set of Boolean variables that encode keystream generator registers’ initial state (see Table 1) was used. This is possible because the corresponding Boolean variables encode the function input, i.e. they can be viewed as the most important. Thus, the search spaces contained $2^{288}$, $2^{160}$, $2^{200}$, $2^{513}$, $2^{512}$ for Trivium, Grain, Mickey, Rabbit and Salsa20, respectively. Further, when it is said that a point has $k$ variables, it means that a corresponding set used to decompose an original SAT instance contains exactly $k$ Boolean variables.

In our experiments to calculate the value of the objective function (1) we used the implementation from the ALIAS tool \[25\]. It is a multi-threaded program based on three ALIAS modules: a Python script alias.py, and two C++ programs sampler and genipainterval. ALIAS operates with incremental SAT solvers via the IPASIR interface \[2\]. In all experiments described below the IPASIR-based version of the sequential rokk SAT solver \[42\] was used because it recently has shown good results on SAT-based cryptanalysis problems.

To perform the experiments, the HPC-cluster “Academician V.M. Matrosov” \[9\] was employed. Each its computational node is equipped with $2 \times 18$-core Intel Xeon E5-2695 CPUs and 128 Gb of RAM. While the implementation of the objective function is a multi-threaded program, at each moment of time an optimization algorithm operates with exactly one point from a search space.

Since SAT is an NP-complete problem and thus the SAT solvers are essentially heuristic algorithms which given a sufficiently hard problem in practice can work for very large amounts of time, it makes sense to restrict the calculation of (1) by introducing the time limit on it. It turned out, that this limit is a very important parameter. The following values of the time limit were considered: no limit, 10 000 seconds, 1 000 seconds, 500 seconds and 100 seconds. While both optimization algorithms implemented in SMAC and ALIAS are stochastic (see Section 3), and the objective function is quite costly, every tool was launched 3 times for 1 day on each configuration (problem, time limit) on one cluster’s computational node (i.e. on 36 CPU cores) to alleviate the
effect of randomness. Thus, $2 \times 3 \times 5 \times 5 = 150$ 1-day launches were performed in total. Note, that only 102 of them resulted in any record point. There are two reasons for this phenomenon. First, SMAC cannot deal with Rabbit and Salsa20 at all, because its maximum available objective function value is lower than baseline estimations for these keystream generators. Apparently, SMAC uses double data type in Java to store the function value, thus when it exceeds $1.7e+308$, its behaviour is undefined. Second, in several cases the values at starting points cannot be calculated due to a low time limit and/or an extremely high cost of the objective function: it occurs on Mickey and Rabbit with the time limit of 100 seconds, and on Salsa20 with the time limit of 100, 500 and 1 000 seconds. For every configuration, the best result (with the minimal objective function value) out of 3 random launches was chosen. The results are shown in Table 2.

<table>
<thead>
<tr>
<th>Time limit (sec)</th>
<th>Tool</th>
<th>Value of objective function</th>
<th>Trivium</th>
<th>[number of variables/total number of variables]</th>
<th>Grain</th>
<th>Mickey</th>
<th>Rabbit</th>
<th>Salsa20</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>SMAC</td>
<td>3.08e+47</td>
<td>3.08e+47</td>
<td>[153/288]</td>
<td>2.66e+45</td>
<td>2.4e+58</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ALIAS</td>
<td>1.73e+44</td>
<td>1.73e+44</td>
<td>[147/288]</td>
<td>4.77e+30</td>
<td>1.56e+48</td>
<td>3.16e+145</td>
<td>5.25e+152</td>
</tr>
<tr>
<td>10 000</td>
<td>SMAC</td>
<td>3.54e+46</td>
<td>3.54e+46</td>
<td>[163/288]</td>
<td>2.92e+38</td>
<td>6.96e+53</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ALIAS</td>
<td>3.62e+44</td>
<td>3.62e+44</td>
<td>[152/288]</td>
<td>4.84e+30</td>
<td>1.6e+50</td>
<td>6.82e+144</td>
<td>5.5e+152</td>
</tr>
<tr>
<td>1 000</td>
<td>SMAC</td>
<td>2.12e+46</td>
<td>2.12e+46</td>
<td>[161/288]</td>
<td>1.98e+33</td>
<td>1.36e+54</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ALIAS</td>
<td>2.46e+41</td>
<td>2.46e+41</td>
<td>[145/288]</td>
<td>4.04e+30</td>
<td>8.18e+50</td>
<td>1.52e+142</td>
<td>-</td>
</tr>
<tr>
<td>500</td>
<td>SMAC</td>
<td>2.47e+46</td>
<td>2.47e+46</td>
<td>[162/288]</td>
<td>7.61e+33</td>
<td>2.69e+54</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ALIAS</td>
<td>1.4e+41</td>
<td>1.4e+41</td>
<td>[144/288]</td>
<td>2.09e+31</td>
<td>2.26e+50</td>
<td>2.57e+139</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>SMAC</td>
<td>6.62e+45</td>
<td>6.62e+45</td>
<td>[161/288]</td>
<td>5.42e+34</td>
<td>4.52e+34</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ALIAS</td>
<td>3.84e+47</td>
<td>3.84e+47</td>
<td>[167/288]</td>
<td>3.75e+32</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

According to the table, ALIAS has shown much better results on the considered problems than SMAC. Also, it was not possible to launch SMAC on Rabbit and Salsa20 due to the reasons discussed above. The time limit had a strong influence. In particular, SMAC did not manage to move from starting points on Grain and Mickey with no time limit, while with the time limit enabled it showed competitive results in several cases. The best results for 3 out of 5 problems were obtained by ALIAS with the time limit enabled: 500 seconds on Trivium (record function value 1.4e+41) and Rabbit (2.57e+139); 1 000 seconds on Grain (4.04e+30). On Mickey and Salsa20 the best results were obtained by ALIAS with no time limit (1.56e+48 and 5.25e+152 respectively). In Figure 1 the progress of those launches that gave the best results on Trivium, Grain and Mickey are shown. Note, that on Trivium and Grain the effect of the jump strategy (see Section 3) can be seen.
Figure 1. Minimization on Trivium, Grain and Mickey
Discussion

In [35, 38] objective functions were defined by two different SAT-based Monte Carlo algorithm. In the present study, the objective function is defined based on an improved SAT-based Monte Carlo algorithm suggested in [43]. In [35, 38] the GBFS without any jump strategy was used. In [25] it was extended by a random jump strategy. In the present study, the GBFS with an improved jump strategy that employs information about the search progress is used.

The considered optimization problems were solved on a computing cluster. Also, they can be solved using the same objective function via cloud computing (e.g., by the Autotuner service [40]) or via distributed computing (e.g., by BNB-solver [14]). Note, that optimization algorithms implemented in these tools do not satisfy the third criterion (see Section 3), that is why we did not use them in the present study. A number of optimization algorithms can be also used to minimize this function, they were mentioned in Section 3.

The idea of the SAT-based cryptanalysis was first proposed in [10]. In [28] a reduced variant of the block cipher DES was analyzed using SAT. In [31] SAT was used to study cryptographic hash functions of the MD family. In [12, 39, 37, 35, 43, 38] several cryptographic keystream generators (Crypto-1, Hitag2, Alternating step generator, A5/1, Trivium, Grain0) were analyzed using the SAT-based cryptanalysis. It should be noted, that four keystream generators (Grain1, Mickey, Rabbit, Salsa20) have not been analyzed by the SAT-based cryptanalysis before the present paper.

From the cryptanalysis viewpoint, our runtime estimations obtained in the previous section for Rabbit and Salsa20 ciphers are not very interesting. Both ciphers are exceptionally resistant to cryptographic attacks in general and to SAT-based cryptanalysis in particular. Since the number of bits that need to be guessed in constructed attacks is comparable with the total number of input bits, it means that the attacks in question are unlikely to be faster than brute force search.

Apparently, the best results for cryptanalysis of Trivium using a small amount of keystream were presented in [20]. That attack employs a similar idea to decompose the original hard problem by instantiating the values of Boolean variables, however, the system of Boolean equations is solved not via SAT but by a so-called Characteristic Set Method. The attack from [20] uses about 190 bits of keystream and has an estimated runtime of $2^{115} \approx 4.15e+34$ seconds, which is about $3e+6$ times better than the attack constructed by ALIAS-based GBFS (see Section 5). It appears that the characteristic set method itself suits better to cryptanalysis of Trivium. The best SAT-based attack on Trivium proposed in the present study (with the estimation of $1.4e+41$ seconds) is slightly better than the previous best such attack, described in [38] ($2.04e+41$).

With regard to cryptanalysis of the Mickey cipher, the state of the art results are the ones obtained in [26]. That paper uses the related key attack technique which, essentially, implies having information about several (sometimes several dozens) keys which are related to each other together with keystream fragments obtained using these keys. Thus it is hard to compare our results with that from [26] since the conditions of the attacks are drastically different. Informally, the attack from [26] requires much more data which is quite hard to obtain. However, in optimal conditions, it allows performing cryptanalysis of Mickey in mere minutes. Our attack requires one small fragment of keystream and no specific knowledge about keys, but the runtime estimation is very large.

The Grain cipher has two modifications, informally denoted as Grain0 and Grain1. The first one represents the initial version of Grain submitted to the e-Stream project. The second one (Grain1) is the variant with resolved vulnerabilities found in Grain0. To the best of our knowledge
there are no known SAT-based attacks on Grain1, thus our results are novel in this direction. The other relevant methods for cryptanalysis of the Grain cipher either rely on combinations of weak secret keys and initialization vectors [44] or require inordinate amounts of keystream [27].

Conclusion

The novelty of the present research is two-fold. First, apparently for the first time, the SAT-based cryptanalysis of several e-STREAM finalists was considered. While it did not lead to any groundbreaking results, it should be noted that such results usually imply quite a thorough and detailed analysis of underlying cryptographic functions and here we use an automated “number-crunching” scheme. The second contribution of the paper consists in comparing existing widely used algorithms for finding decompositions for hard SAT instances with state-of-the-art tools for blackbox optimization, represented by SMAC tool.

Overall, we believe that the present direction of research is promising and can be used to solve hard SAT instances both related and unrelated to cryptography.

Acknowledgements

The research was partially supported by Council for Grants of the President of the Russian Federation grant no. MK-4155.2018.9, RFBR grant no. 16-07-00155-a. Stepan Kochemazov was additionally supported by Council for Grants of the President of the Russian Federation (stipend no. SP-1829.2016.5).

References

[7] Christophe De Cannière and Bart Preneel. Trivium. In Robshaw and Billet 33, pages 244–266.


