ALIAS: A Modular Tool for Finding Backdoors for SAT

Stepan Kochemazov and Oleg Zaikin

ISDCT SB RAS, Irkutsk, Russia
veinamond@gmail.com, zaikin.icc@gmail.com

Abstract. We present ALIAS, a modular tool aimed at finding backdoors for hard SAT instances. Here by a *backdoor* for a specific SAT solver and SAT formula we mean a set of its variables, all possible instantiations of which lead to construction of a family of subformulas with the total solving time less than that for an original formula. For a particular backdoor, the tool uses the Monte-Carlo algorithm to estimate the runtime of a solver when partitioning an original problem via said backdoor. Thus, the problem of finding a backdoor is viewed as a black-box optimization problem. The tool's modular structure allows to employ state-of-the-art SAT solvers and black-box optimization heuristics. In practice, for a number of hard SAT instances, the tool made it possible to solve them much faster than using state-of-the-art multithreaded SAT-solvers.

1 Introduction

Informally, a *backdoor* is some hidden flaw in a design of a system that allows one to do something within that system that should not be possible otherwise. In the context of Constraint Satisfaction Problems (CSP) a backdoor is usually a small subset of problem variables which has a peculiar property: instantiating backdoor variables results in a subproblem that is significantly easier to solve. For the first time the concept of backdoors arose in the context of CSP in [26], where *strong backdoors* were introduced and analyzed. Their main disadvantage is that they rely on polynomial algorithms to solve simplified subproblems, and thus strong backdoors that can be used in practice are very hard to find [15, 23].

In the present paper, we consider more general backdoors to SAT, that do not rely on polynomial algorithms to solve simplified subproblems. In particular, we search for such sets of variables of a considered SAT instance that all possible instantiations of backdoor variables results in a family of subproblems, for which a total solving time is less than that for an original SAT instance. It is clear that such subproblems can be solved in parallel. For a given SAT instance C, solver S and backdoor B one can effectively compute the estimation of runtime of S on a family of subproblems produced by assigning values to variables from B in C using a Monte-Carlo method. Thus there is defined a black-box pseudo-Boolean function with aforementioned inputs. Then, it is possible to use arbitrary black-box pseudo-Boolean optimization methods to traverse the search space of possible general backdoors to find one with a good estimation.

We implemented this approach in the form of *modulAr tooL for fInding bAckdoors for Sat* (ALIAS) – a convenient customizable scalable tool that can employ arbitrary incremental state-of-the-art SAT solvers and black-box optimization heuristics to search

for backdoors to hard SAT instances. The found backdoor is then used to solve the corresponding instance by the same incremental solver. Thereby, ALIAS can be viewed as a tool for constructing backdoor-based divide-and-conquer parallel SAT solvers. The ALIAS tool and our benchmarks are available at https://github.com/Nauchnik/alias.

2 On Backdoors to SAT

Suppose C is a SAT instance, X is a set of its variables, and A is a polynomial algorithm. If we assign values $\alpha = (\alpha_1, \ldots, \alpha_k)$ to variables from set B,$|B| = k$, $B \subseteq X$, then the simplified SAT instance is denoted as $C[\alpha/B]$.

Definition 1 (Backdoor [26]). *A nonempty subset B of variables from C is called a* backdoor *in C for A if for some instantiation β of variables from B an algorithm A returns a satisfying assignment of $C[\beta/B]$.*

Note that the definition of backdoor implies only satisfiable instances and can not be easily extended to unsatisfiable ones. Also, even if backdoor is known, it is necessary to find such β that A would be able to solve a considered instance.

Definition 2 (Strong Backdoor [26]). *A nonempty subset B of variables from C is a* strong backdoor *in C for A if for any instantiation γ of variables from B an algorithm A returns a satisfying assignment or concludes unsatisfiability of $C[\gamma/B]$.*

For SAT instances the natural choice of polynomial algorithm A is the Unit Propagation rule (UP) [8]. A Strong Backdoor w.r.t UP is called Strong Unit Propagation Backdoor Set (SUPBS). For any SAT instance the whole set of its variables is a SUPBS (further it is called *trivial SUPBS*). If a SAT instance encodes a Boolean circuit, a set of variables encoding its input can usually serve as a SUPBS.

Compared to a backdoor a strong backdoor is much more powerful: given a strong backdoor B, one can traverse through possible instantiations of variables from B thus solving C in time $\approx 2^{|B|} \times |C|$ (here $|C|$ is the size of a SAT instance C in computer memory). However, it is unclear what to do if, for example, $|B| > 100$. Also, the problem of finding strong backdoors for SAT is particularly hard, see e.g. [15].

The main disadvantage of the notion of strong backdoor lies in polynomial complexity requirement for an algorithm used to solve constructed subproblems. The following definition in a way extends the notion of backdoors to non-polynomial algorithms. Assume that G is an arbitrary complete SAT solving algorithm.

Definition 3 (Non-deterministic Oracle Backdoor Set (NOBS) [21]). *A non-empty set B of variables from C is a* Non-deterministic Oracle Backdoor Set (NOBS) *w.r.t. algorithm G if the total running time of G given formulas $C[\beta/B]$, $\beta \in \{0,1\}^{|B|}$, is less than the running time of G on the original formula C.*

Without formally defining NOBS, the corresponding idea was used in a number of papers on application of SAT to cryptanalysis instances, such as [7, 9, 22, 27]. Compared to strong backdoors, NOBS do not give a straightforward way to estimate the runtime of G for solving C using backdoor B. However, it can be done via the Monte-Carlo method [17] as follows. We treat the average runtime of G on an arbitrary subproblem

$C[\gamma/B]$, $\gamma \in \{0,1\}^{|B|}$ as a random variable. The intermediate goal is to estimate its expected value. For this purpose, first, construct a random sample of size N of instantiations of variables from B:$\{\beta_1, \ldots, \beta_N\}$, $\beta_i \in \{0,1\}^{|B|}$, $i \in \{1, \ldots, N\}$. Second, measure the runtime of G on $C[\beta_i/B]$, $i \in \{1, \ldots, N\}$, denote it by $T_G(\beta_i)$. Then the runtime estimation can be computed using the formula:

$$Runtime\_Estimation(C, B, G, N) = 2^{|B|} \times \frac{1}{N} \times \sum_{i=1}^{N} T_G(\beta_i) \tag{1}$$

Since G is a complete algorithm, theoretically, the value of $Runtime\_Estimation$ function can be computed for any B. Essentially, it is a blackbox function. One possible way to find a good backdoor B is for fixed C, G and N to minimize the value of $Runtime\_Estimation(C, B, G, N)$ by varying B. Since any backdoor B can be uniquely represented by a Boolean vector from $\{0,1\}^{|X|}$, where X is the set of variables from C, it means that the corresponding search space is $\{0,1\}^{|X|}$.

3 The ALIAS tool

Essentially, the ALIAS tool implements the blackbox optimization in the space of possible NOBS. The blackbox function in ALIAS is computed according to (1). The flowchart of the tool is presented in Figure 1.

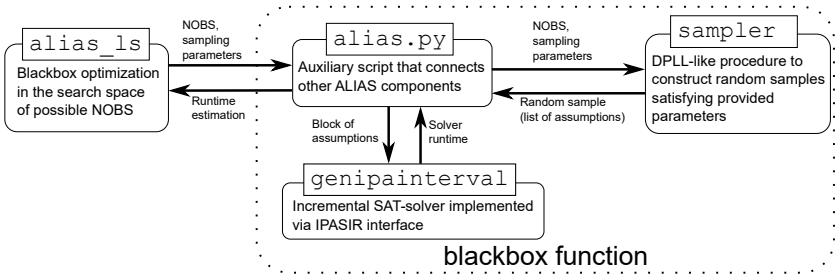


Fig. 1: ALIAS flowchart

ALIAS consists of four interconnected modules: ALIAS_LS, ALIAS.PY, SAMPLER and GENIPAINTERVAL program. The latter three implement the aforementioned blackbox function which for a given incremental SAT solver, SAT instance and NOBS computes a runtime estimation for this instance. Detailed comments are presented below.

ALIAS_LS module Note that due to the fact that the search space of possible NOBS in the general case is extremely large, any possible way to restrict it is welcomed. Because of this, in ALIAS the search space of possible NOBS always consists of possible subsets of a SUPBS, either trivial or nontrivial.

Now, assume that a SUPBS for a considered SAT instance contains N variables. Then the search space has 2^N points, each point corresponding to some NOBS. For each NOBS we can compute the runtime estimation using (1) (for a fixed SAT solver). So the goal of ALIAS_LS is to traverse the search space towards NOBSs with minimal runtime estimations. Currently, for this purpose ALIAS_LS uses a simple optimization algorithm based on the Greedy Best First Search (GBFS) [19]. GBFS uses SUPBS as a starting point to construct a baseline runtime estimation. Then it 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 (1) for an arbitrary point is quite costly, all passed points are stored in order to avoid recomputing (1) for corresponding backdoors.

The GBFS implementation in ALIAS uses two additional heuristics. First, at the beginning of the search the algorithm tries to quickly traverse the search space by removing large amount of randomly chosen variables (10 in our experiments) from the current record point at each step as long as it leads to updating the record. It often allows to quickly move from NOBS with hundreds of variables to that with dozens. The second heuristic is that when a local minimum is reached, the algorithm tries to jump from it by constructing a new starting point by permuting the current record point. The algorithm stops either if the time limit is exceeded, if the limit of jumps is reached or if the current runtime estimation is lower than the (scaled) remaining time.

On the current stage the ALIAS components are configured in a way to support optimization tools, which were used in Configurable SAT Solver Competition (CSSC) 2013 and 2014 [14], such as ParamILS [13], SMAC [12], and GGA [1]. Similar to our implementation, all these tools make use of the .pcs file that contains Boolean variables corresponding to the starting point (SUPBS).

ALIAS.PY module The ALIAS.PY is an auxiliary Python 3.6 script that ties together other ALIAS components. It launches and controls all computations, processes the data from SAMPLER and GENIPAINTERVAL, constructs the runtime estimation for a given SAT instance, solver and NOBS, thus implementing a blackbox function. It can also be used to solve a SAT instance using the provided NOBS. In all modes ALIAS.PY can employ multiple CPU cores.

When constructing a runtime estimation, ALIAS.PY implements the Monte-Carlo method: it uses SAMPLER to construct a random sample of subproblems (in the form of assumptions for a SAT solver), then gives them in blocks of fixed size to GENIPAINTERVAL solver (by a *block* we mean a set of instantiations of backdoor variables, in form of assumptions), computes the average solving time for an arbitrary subproblem from random sample, uses it to compute runtime estimation and returns it to ALIAS_LS.

In the solving mode ALIAS.PY splits all possible instantiations of a provided NOBS variables into small blocks and feeds them to GENIPAINTERVAL until either all blocks are processed or a satisfying assignment is found.

SAMPLER module SAMPLER is a program for generating random samples that is implemented on the basis of COMINISATPS solver [18]. Generally speaking, a random

sample can be constructed in many different ways. In the most simple case for a NOBS B we can simply take N randomly generated vectors from $\{0,1\}^{|B|}$ and view them as a random sample by assigning corresponding values to variables from B. This approach was used in [9, 20, 22]. However, the described straightforward sampling procedure might not benefit fully from the incremental solving ability of state-of-the-art SAT solvers because the assignments of variables are too distant from each other (for example Hamming distance-wise). Thus, by default SAMPLER uses the sampling strategy proposed in [27]. Informally, it attempts to construct sequences of backdoor instantiations which are close to each other as nodes of the search tree. At the same time SAMPLER when possible puts into a random sample only subproblems that can not be solved using UP.

GENIPAINTERVAL module The GENIPAINTERVAL program, given a CNF formula and a set of assumptions processes the latter sequentially in incremental way. To build it one needs the IPASIR API [4] and sources of some generic IPASIR-compatible incremental SAT solver. It is natural to consider only incremental solvers since the subproblems produced by instantiating NOBS variables are very similar to each other. Currently, different GENIPAINTERVAL instances running in parallel are not configured to share any information.

4 Experimental results

In all experiments described below we employed one node of the HPC-cluster "Academician V.M. Matrosov" [1] (2 $\times$ 18-core Intel Xeon E5-2695 CPUs and 128 Gb of RAM). Each considered solver was launched on 1 node with 36 threads.

We benchmarked ALIAS against the Top 3 solvers from the SAT Competition 2017 Parallel track: SYRUP [3], PLINGELING [5] and PAINLESS-MAPLECOMSPS [10]. All these solvers are portfolio. As IPASIR-based solvers for ALIAS we used the Top 3 solvers from the SAT Competition 2017 Incremental track: ABCDSAT [6], GLUCOSE [2] and RISS [16]. The resulting parallel solvers are denoted as ALIAS-ABCDSAT, ALIAS-GLUCOSE and ALIAS-RISS.

In preliminary experiments we compared the effectiveness of GBFS implementation in ALIAS_LS with that of SMAC tool [12] as blackbox heuristics for finding NOBS. For all considered instances GBFS found backdoors with better runtime estimation, thus it was used in all experiments below.

Each ALIAS solver works as follows: first ALIAS_LS is launched with a specified time limit. Once it found a good NOBS or exceeded the time limit, the best found NOBS is then used to solve the instance for the remaining time (if any) by instantiating backdoor variables and solving corresponding subproblems in parallel.

Two benchmark sets of hard SAT instances were considered. The first set consists of instances, in which a relatively small SUPBS is known. It is formed by SAT encodings of cryptanalysis instances for the alternating step generator (ASG) [11] and two its modifications, MASG and MASG0 [25]. SAT instances for these problems were

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

taken from [27]: 10 for each of ASG-72, ASG-96, MASG-72 and MASG0-72 (40 in total). Naturally, for ASG-72, MASG-72 and MASG0-72 there is a SUPBS of 72 variables and for ASG-96 a SUPBS of 96 variables (corresponding to secret keys). Thus, ALIAS-based solvers were provided with this information. Each instance from this set has exactly one satisfying assignment.

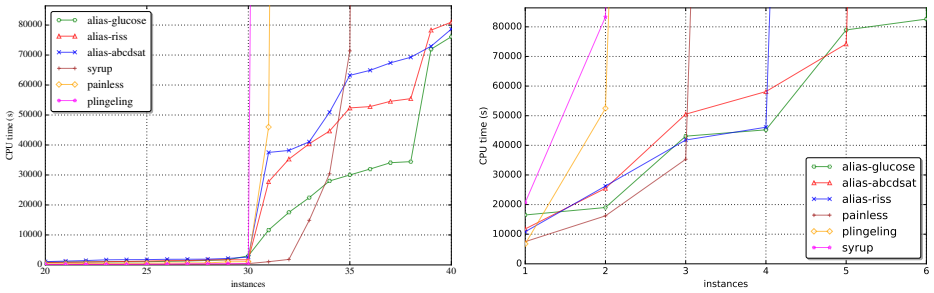
The second benchmark set contains hard small crafted SAT instances. To construct it we first took all crafted instances with less than 500 variables from SAT Competitions 2007, 2009, 2011, 2014, 2016, 2017 and also *challenge-105.cnf* described in [24]. Then we launched SYRUP, PLINGELING and PAINLESS-MAPLECOMSPS on each of them with the time limit of 5000 seconds. It turned out that 33 instances were not solved in time by any solver: 7 from SAT Competition 2007, 10 from SAT Competition 2009, 9 from SAT Competition 2011, 6 from SAT Competition 2014 and also *challenge-105.cnf*. Thus, these 33 instances form the second benchmark set. For the instances from the second benchmark set the ALIAS-based solvers were given only a trivial SUPBS – the whole set of variables of a corresponding formula.

The 6 considered solvers were launched on two described sets (73 instances in total) with the time limit of 1 day. The obtained results are presented in Figures 2a and 2b. Note, that 26 out of 33 instances from the second benchmark set were not solved within the time limit by any considered solver. Table 1 lists the instances from the second set, which were solved within the time limit by at least one solver. This table also contains data on found backdoors. For ASG-72, ASG-96, MASG-72 and MASG0-72 the information is presented for 1 instance out of 10, the results for other instances from the series are similar. Here $|B|$ is a size of a found backdoor, BT – time spent to find it, RE – its runtime estimation (1), ST – the solving time using the found backdoor.

Table 1: Data on found backdoors. RE, BT, ST – time in seconds.

instance	**alias-glucose**				**alias-abcd**				**alias-riss**			
	$\|B\|$	RE	BT	ST	$\|B\|$	RE	BT	ST	$\|B\|$	RE	BT	ST
ASG-72-0	23	365	412	432	15	390	1713	210	20	308	330	103
MASG-72-0	19	347	348	7	19	330	443	44	19	380	585	38
MASG0-72-0	22	417	503	361	18	1167	2195	397	22	723	882	548
ASG-96-0	26	34548	13270	9177	26	36704	22605	42300	27	37661	12872	42583
mod4-2-9-u2	29	1.4e+6	86400	-	28	1.3e+5	19059	-	31	1.8e+5	24510	-
sgen1-sat-250	31	1.7e+5	15740	63220	33	3.2e+5	9379	-	29	1.8e+6	86400	-
sgen6-1200-5	26	1045	1503	14990	26	3684	3685	8092	24	2045	2050	5431
sgen6-1320-5	28	13974	4299	40934	28	16068	4623	45897	29	23388	3281	42833
sgen6-1440-5	29	39239	11896	70725	30	62516	18079	56168	31	144779	13092	-
sgen3-n240	31	1.0e+5	5505	37590	27	17504	7120	51026	30	73406	10341	31451
challenge-105	24	6121	6234	12780	25	3414	3422	22069	26	4840	4218	22033

On the first benchmark set the ALIAS-based solvers greatly outperform the competitors. We also tested ALIAS on ASG tests with trivial SUPBS provided (instead of much smaller nontrivial one) and it yielded much worse results. Hence, the knowledge of a nontrivial SUPBS is a big advantage. On the second set the situation is more



(a) ASG-72, ASG-96, MASG-72, MASG0-72 (b) Hard small crafted instances

Fig. 2: Comparison of 3 ALIAS-based solvers with the Top 3 solvers from the SAT Competition 2017 Parallel track

complex: there are instances which are solved by ALIAS-based solvers but not by the competitors and vice versa. Among crafted instances only *sgen3-n240-s78945233-sat* and *sgen1-sat-250-100* are satisfiable.

It should be noted, that strictly speaking, the blackbox optimization procedure employed in ALIAS does not guarantee that the found backdoors are really NOBS (see Section 2). It turned out, that for ASG-72, MASG-72 and MASG0-72 only few found backdoors are NOBS. A possible reason for this is that these instances are quite simple even for sequential solvers. Nevertheless, ALIAS-based solvers and their parallel competitors showed comparable results on them. Meanwhile, for all ASG-96 instances the found backdoors are NOBS, and here ALIAS-based solvers are clear winners. Note, that in Figure 2a values from 31 to 40 on the x-axis correspond to the ASG-96 instances. In the second benchmark set, for *sgen6-1200-5-1* and *challenge-105* the found backdoors are indeed NOBS. For the remaining instances it was impractical to check it, because it would take up to several weeks per instance.

5 Conclusion

The experiments show that the approach to solving hard SAT instances based on sampling, while not a silver bullet, clearly has its applications. We believe that the presented ALIAS tool may be useful in the study of hard SAT instances and sometimes can shed the light on some aspects of their inner structure undetectable by state-of-the-art SAT solvers.

Acknowledgements. We thank anonymous reviewers for their insightful comments that made it possible to significantly improve the quality of presentation.

The research was funded by Russian Science Foundation (project No. 16-11-10046). Stepan Kochemazov was additionally supported by Council for Grants of the President of the Russian Federation (stipend no. SP-1829.2016.5).

References

1. Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: CP. LNCS, vol. 5732, pp. 142–157 (2009)
2. Audemard, G., Lagniez, J., Simon, L.: Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In: SAT. LNCS, vol. 7962, pp. 309–317 (2013)
3. Audemard, G., Simon, L.: Lazy clause exchange policy for parallel SAT solvers. In: SAT. LNCS, vol. 8561, pp. 197–205 (2014)
4. Balyo, T., Biere, A., Iser, M., Sinz, C.: SAT race 2015. Artif. Intell. 241, 45–65 (2016)
5. Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT competition 2017. In: Proc. of SAT Competition 2017. vol. B-2017-1, pp. 14–15 (2017)
6. Chen, J.: Improving abcdSAT by At-Least-One recently used clause management strategy. CoRR abs/1605.01622 (2016), http://arxiv.org/abs/1605.01622
7. Courtois, N.: Low-complexity key recovery attacks on GOST block cipher. Cryptologia 37(1), 1–10 (Jan 2013)
8. 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)
9. Eibach, T., Pilz, E., Völkel, G.: Attacking Bivium using SAT solvers. In: SAT. LNCS, vol. 4996, pp. 63–76 (2008)
10. Frioux, L.L., Baarir, S., Sopena, J., Kordon, F.: PaInleSS: A framework for parallel SAT solving. In: SAT. LNCS, vol. 10491, pp. 233–250 (2017)
11. Günther, C.G.: Alternating Step Generators Controlled by De Bruijn Sequences, LNCS, vol. 304, pp. 5–14 (1988)
12. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Proc. of LION-5. p. 507523 (2011)
13. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. JAIR 36, 267–306 (2009)
14. Hutter, F., Lindauer, M., Balint, A., Bayless, S., Hoos, H., Leyton-Brown, K.: The configurable SAT solver challenge (CSSC). Artificial Intelligence 243, 1 – 25 (2017)
15. Kilby, P., Slaney, J.K., Thiébaux, S., Walsh, T.: Backbones and backdoors in satisfiability. In: AAAI 2005. pp. 1368–1373 (2005)
16. Manthey, N.: Towards next generation sequential and parallel SAT solvers. Constraints 20(4), 504–505 (2015)
17. Metropolis, N., Ulam, S.: The Monte Carlo Method. J. Amer. statistical assoc. 44(247), 335–341 (1949)
18. Oh, C.: Between SAT and UNSAT: the fundamental difference in CDCL SAT. In: SAT. LNCS, vol. 9340, pp. 307–323 (2015)
19. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 3rd edn. (2009)
20. 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)
21. Semenov, A., Zaikin, O., Otpuschennikov, I., Kochemazov, S., Ignatiev, A.: On cryptographic attacks using backdoors for SAT (in print). In: AAAI 2018 (2018)
22. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: SAT. LNCS, vol. 5584, pp. 244–257 (2009)
23. Szeider, S.: Backdoor sets for DLL subsolvers. Journal of Automated Reasoning 35(1), 73–88 (2005)
24. Van Gelder, A., Spence, I.: Zero-one designs produce small hard SAT instances. In: SAT. LNCS, vol. 6175, pp. 388–397 (2010)

25. Wicik, R., Rachwalik, T.: Modified alternating step generators. IACR Cryptology ePrint Archive 2013, 728 (2013)
26. Williams, R., Gomes, C.P., Selman, B.: Backdoors to typical case complexity. In: IJCAI. pp. 1173–1178 (2003)
27. Zaikin, O., Kochemazov, S.: An improved SAT-based guess-and-determine attack on the alternating step generator. In: ISC 2017. LNCS, vol. 10599, pp. 21–38 (2017)