CryptoMiniSat 2.5.1

Mate Soos
LIP6 UPMC, SALSA team INRIA Rocquencourt, PLANETE team INRIA Grenoble

I. Introduction

In this solver description we present the feature-set of CryptoMiniSat, a modern SAT Solver that aims to unify the advantages of SatELite [1], PrecoSat [2], GLUCOSE [3] and MiniSat [4] with the xor-clause handling of version 1 of CryptoMiniSat [5] to create a formula that can solve many types of different problem instances under reasonable time.

II. Features

CryptoMiniSat is a DPLL-based SAT solver developed from MiniSat. The following list of non-exhaustive features are offered by CryptoMiniSat relative to the original "core" MiniSat.

A. Xor clauses

XOR clauses are extracted at the beginning of the solving. They are subsequently treated differently. They have their own watchlists, their own propagation mechanism, and their own subsumption algorithm. This should mean that they are handled faster in most scenarios.

B. Binary xor clauses

Binary xor clauses are handled specially. Firstly, they are regularly searched for using a special heuristic. Secondly, a forest structure is built from them, indicating which variable is equi- or antivalent with which variable. The top of the trees are regularly replaced with those lower in the tree, reducing the number of clauses and variables in the problem, and usually leading to variable assignments (and possibly even more binary xor clauses).

C. Binary xor clause finding through regular XOR-ing of xor clauses

As per the PhD Thesis of Heule [6], xor clauses are regularly XOR-ed with one another to obtain different XOR clauses. However, contrary to that present in the paper, the smaller XOR-s are only acted upon if they are binary. In this case, they are added to the forest of equi- and antivalences, and replaced with one another at a later time, according to a heuristic.

D. Phase calculation, saving and random flipping

Default phase is calculated for each variable according to the Jeroslow and Wang [7] heuristic. The phases are saved, according to Pipatsrisawat and Darwiche [8]. The phase, however, is randomly flipped at intervals that is determined by the problem. The average branch depth is measured, and with P(1/avgBranchDepth), the current phase is flipped. According to our experience, this helps in exploring new places in the search space.

E. Automatic detection of cryptographic and industrial instances

Industrial and cryptographic instances are very different. They need different restart strategies and they need different learnt clause activity statistics. We try to detect which problem belongs to which family, and use GLUCOSE-style learnt clause heuristics [9] or MiniSat-style learnt clause activity accordingly. We also switch the restart type from dynamic to static and vica-versa. The detection is based on the percentage of xor clauses and the stability of variable activity. Either of the two is too high, the problem is deemed to be cryptographic. The stability of variable activity is measured through saving of the top 100 variables, and comparing them with the next restart's top 100 variables. This is done for 5 restarts, and at the end, the decision is made. The detection routine is run regularly, to detect whether the problem has changed enough to switch from one type to the other.

F. Variable elimination, clause subsumption and clause strengthening

SatELite-type variable elimination, clause subsumption and clause strengthening is regularly performed. The occurrence lists are, however, not updated all the time such as the case with other solvers. Instead, occurrences are calculated on per-use basis. The number of variable elimination cycles, clause subsumption cycles and clause strengthening cycles are limited each time the simplification is done such as to avoid the routine taking overly large amounts of time.

G. On-the-fly clause improvement

Since the occurrence lists are not updated all the time, the only way to carry out subsumption is the algorithm by Han and Somenzi [10]. This lightweight subsumption-check is carried out every time a conflict analysis is done.

H. Binary clause propagation

Binary clauses are in a separate watchlist, as per GLUCOSE [3]. They are fully propagated before other clauses are propagated. The propagation order is: binary clauses, regular clauses, xor clauses. As per PrecoSat [2], the binary clauses are always fully propagated, regardless if a conflict has been found earlier. The conflict analysis routine is then called on the last conflicting binary clause.

I. 32-bit pointers on 64-bit architectures under Linux

64-bit pointers are well-known to slow down the solving of SAT solvers, due to the extra memory and thus cache space occupied by them when going through the watchlists in the propagation phase. This limitation means that all code has to be compiled as 32-bit code, which means that

extra registers and instructions provided by modern 64-bit architectures is lost. We counter this phenomenon with small pointers. Since the memory used by SAT solvers is rarely more than 4GB, the pointers rarely contain more than 32 bit real information. We extract this information, and only store these 32 bits.

J. Binary graph treatment

Binary clauses generated by hyper-binary resolution [11] are added in an optimal manner: the binary subtree of literal a is searched and the highest-degree dominated literal c still leading to b is connected to b. This ensures maximal graph connectivity and sparsity. Binary clauses describing tautologies such as $(\neg a \vee b), (\neg b \vee c), (\neg \mathbf{a} \vee \mathbf{c})$ are regularly removed. Tautologies are also regularly and temporarily generated to subsume and strengthen other clauses.

K. Clause cleaning

Clauses are regularly removed that have at least one of their literals assigned to `true`. Contrary to "core" MiniSat, we also remove `false` literals from clauses, shortening them. Interestingly, this does not need these clauses to be re-attached, as `false` literals are not in the watchlists — or if they are, the clause is satisfied, and can be fully removed.

L. Sub-problem detection and handling

Problems can sometimes contain multiple fully distinct sub-problems. We build a connection graph between clauses, and treat graph components as distinct problems. These sub-problems are solved with sub-solvers, their solutions are saved, and finally added at the solution extension phase. If any of the sub-problems are UNSAT, then the whole problem is UNSAT. Interestingly, it is a good idea to check for sub-problems regularly. As problems are solved, variables are sometimes assigned at decision level zero, disconnecting the graph into distinct components. We regularly check for such occurrences, and solve the sub problems as described.

M. Xor clause subsumption

Xor clauses can be subsumed similarly to normal clauses. Since xor clauses represent many regular clauses, doing the subsumption natively saves significant time.

N. Dependent variable removal

Dependent variables , as per [6] are removed along with their corresponding xor clause. Dependent variables are variables that appear nowhere else but in exactly one xor clause. Since that xor clause can always be satisfied by a correct value of the dependent variable, the xor clause can be removed without further ado, and reintroduced during solution extension as per SatELite. This removes a constraint and a variable from the problem. Note that this variable could not have been removed as part of pure literal elimination. However, interestingly, blocked clause elimination (BCE) can remove these clauses and corresponding variable(s). This connection has not been noted by Biere and Jarvisalo in [12], but shows the effectiveness of their method.

O. Failed literal probing

Variables are tried to be branched both to `true` and `false` at regular intervals. If any of the branches fails (conflict is returned), that variable is assigned to the other branching. Otherwise, the assignments of both are saved and compared with one another. If they contain a common subset, that variable is assigned, as per [13]. An interesting addition to this is the method by Li [14], where binary XOR clauses are found in the same way that common subset of assignments are found. Binary xor clause $l \oplus u$ is also found when $u \in Prop(clauses, l)$ and $\neg u \in Prop(clauses, \neg l)$, following Proposition 4 of [13].

Acknowledgements

The author was supported by the RFID-AP Projet of ANR, project number `ANR-07-SESU-009`.

I would like to thank Martin Maurer, Trevor Hansen and Vijay Ganesh for their bug reports, ideas and stress-tests.

Experiments carried out to tune CryptoMiniSat were performed using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies [15].

References

[1] Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. [17] 61–75

[2] Biere, A.: P{re,i}cosat@sc'09: a solver that predicts learnt clauses quality. In: SAT 2009 competitive events booklet. (2009) 41–42

[3] Audemard, G., Simon, L.: GLUCOSE: a solver that predicts learnt clauses quality. In: SAT 2009 competitive events booklet. (2009) 7–8

[4] Eén, N., Sörensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: SAT. Volume 2919 of LNCS., Springer (2003) 502–518

[5] Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. [16] 244–257

[6] Heule, M.J.: Smart solving: Tool and techniques for satisfiability solvers. Technical report, Technische Universiteit Delft (2008)

[7] Jeroslow, R.G., Wang, J.: Solving propositional satisfiability problems. Ann. Math. Artif. Intell. **1** (1990) 167–187

[8] Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In Marques-Silva, J., Sakallah, K.A., eds.: SAT. Volume 4501 of Lecture Notes in Computer Science., Springer (2007) 294–299

[9] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In Boutilier, C., ed.: IJCAI. (2009) 399–404

[10] Han, H., Somenzi, F.: On-the-fly clause improvement. [16] 209–222

[11] Gershman, R., Strichman, O.: Cost-effective hyper-resolution for preprocessing CNF formulas. [17] 423–429

[12] Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In Esparza, J., Majumdar, R., eds.: TACAS. Volume 6015 of Lecture Notes in Computer Science., Springer (2010) 129–144

[13] Berre, D.L.: Exploiting the real power of unit propagation lookahead. Electronic Notes in Discrete Mathematics **9** (2001) 59–80

[14] Li, C.M.: Equivalent literal propagation in the DLL procedure. Discrete Applied Mathematics **130**(2) (2003) 251–276

[15] The Grid'5000 team: The Grid'5000 project https://www.grid5000.fr.

[16] Kullmann, O., ed.: Theory and Applications of Satisfiability Testing — SAT 2009, Swansea, UK. In Kullmann, O., ed.: SAT. Volume 5584 of LNCS., Springer (2009)

[17] Bacchus, F., Walsh, T., eds.: Theory and Applications of Satisfiability Testing, St. Andrews, UK. In Bacchus, F., Walsh, T., eds.: SAT. Volume 3569 of LNCS., Springer (2005)