The GRAT Tool Chain

Efficient (UN)SAT Certificate Checking with Formal Correctness Guarantees

Peter Lammich

Technische Universität München, lammich@in.tum.de

Abstract. We present the GRAT tool chain, which provides an efficient and formally verified SAT and UNSAT certificate checker. It utilizes a two phase approach: The highly optimized gratgen tool converts a DRAT certificate to a GRAT certificate, which is then checked by the formally verified gratchk tool.
On a realistic benchmark suite drawn from the 2016 SAT competition, our approach is faster than the unverified standard tool drat-trim, and significantly faster than the formally verified LRAT tool. An optional multithreaded mode allows for even faster checking of a single certificate.

1 Introduction

The complexity and high optimization level of modern SAT solvers makes them prone to bugs, and at the same time hard to (formally) verify. A common approach in such situations is certification, i. e. to make the SAT solver produce a certificate for its output, which can then be checked independently by a simpler algorithm. While SAT certificates describe a satisfying assignment and are straightforward to check, UNSAT certificates are more complex. The de facto standard are DRAT certificates [15] checked by drat-trim [3]. However, efficiently checking a DRAT certificate still requires a quite complex and highly optimized implementation.[1] A crucial idea [2] is to split certificate checking into two phases: The first phase produces an enriched certificate, which is then checked by the second phase. This effectively shifts the computationally intensive and algorithmically complex part of checking to the first phase, while the second phase is both computationally cheap and algorithmically simple, making it amenable to formal verification.

Cruz-Filipe et al. [2] originally implemented this approach for the weaker DRUP certificates [14], and later extended it to DRAT certificates [1,6], obtaining the LRAT tool chain. Independently, the author also extended the approach to DRAT certificates [9]. While Cruz-Filipe et al. use an extended version of drat-trim to enrich the certificates, the author implemented the specialized gratgen tool for that purpose. Compared to drat-trim, gratgen's distinguishing feature is its support for multithreading, allowing it to generate enriched certificates several times faster at the cost of using more CPU time and memory. Moreover, we have

[1] We found several bugs in drat-trim. Most of them are already fixed [4, 9].

implemented some novel optimizations, making gratgen faster than drat-trim even in single-threaded mode. While [9] focuses on the formal verification of the certificate checker (gratchk), this paper focuses on gratgen. The GRAT tools and raw benchmark data are available at `http://www21.in.tum.de/~lammich/grat/`.

2 The GRAT Toolchain

To obtain a formally verified solution for a CNF formula, it is first given to a SAT solver. If the formula is satisfiable, the SAT-solver outputs a valuation of the variables, which is then used by gratchk to verify that the formula is actually satisfiable. If the formula is unsatisfiable, the SAT-solver outputs a DRAT certificate. This is processed by gratgen to produce a GRAT certificate, which, in turn, is used by gratchk to verify that the formula is actually unsatisfiable. We have formally proved that gratchk only accepts satisfiable/unsatisfiable formulas.

2.1 DRAT Certificates

A DRAT certificate [15] is a list of clause addition and deletion items. Clause addition items are called *lemmas*. The following pseudocode illustrates the *forward checking* algorithm for DRAT certificates:

```
F := F₀  // F₀ is CNF formula to be certified as UNSAT
F := unitprop(F); if F == conflict then exit "s UNSAT"

for item in certificate do
  case item of
    delete C => F := remove_clause(F,C)
  | add C =>
      if not hasRAT(C,F) then exit "s ERROR Lemma doesn't have RAT"
      F := F ∧ C
      F := unitprop(F); if F == conflict then exit "s UNSAT"

exit "s ERROR Certificate did not yield a conflict"
```

The algorithm maintains the invariant that F is satisfiable if the initial CNF formula F_0 is satisfiable. Deleting a clause and unit propagation obviously preserve this invariant. When adding a clause, the invariant is ensured by the clause having the RAT property. The algorithm only reports UNSAT if F has clearly become unsatisfiable, which, by the invariant, implies unsatisfiability of F_0. A clause C has the *RAT property* wrt. the formula F iff there is a *pivot literal* $l \in C$, such that for all *RAT candidates* $D \in F$ with $\neg l \in D$, we have $(F \wedge \neg(C \cup D \setminus \{\neg l\}))^{\mathrm{u}} = \{\emptyset\}$. Here, F^{u} denotes the unique result of unit propagation, where we define $F^{\mathrm{u}} = \{\emptyset\}$ if unit propagation yields a conflict. Exploiting that $(F \wedge \neg(C \cup D))^{\mathrm{u}}$ is equivalent to $((F \wedge \neg C)^{\mathrm{u}} \wedge \neg D)^{\mathrm{u}}$, the candidates do not have to be checked if the first unit propagation $(F \wedge \neg C)^{\mathrm{u}}$ already yields a conflict. In this case, the lemma has the *RUP property*. This optimization is essential, as most lemmas typically have RUP, and gathering the list of RAT candidates is expensive.

2.2 GRAT Certificates

The most complex and expensive operation in DRAT certificate checking is unit propagation,[2] and highly optimized implementations like two watched literals [11] are required for practically efficient checkers. The main idea of enriched certificates [2] is to make unit propagation output a sequence of the identified unit and conflict clauses. The enriched certificate checker simulates the forward checking algorithm, verifying that the clauses from the certificate are actually unit/conflict, which is both cheaper and simpler than performing fully fledged unit propagation. For RAT lemmas, the checker also has to verify that all RAT candidates have been checked.

A GRAT certificate consists of a lemma and a proof part. The lemma part contains a list of lemmas to be verified by the forward checking algorithm, and the proof part contains the unit and conflict clauses, deletion information, and counters how often each literal is used as a pivot in a RAT proof.

The lemma part is stored as a text file roughly following DIMACS CNF format, and the proof part is a binary file in a proprietary format. The gratchk tool completely reads the lemmas into memory, and then streams over the proof during simulating the forward checking algorithm. We introduced the splitting of lemmas and proof after gratchk ran out of memory for some very large certificates.[3]

2.3 Generating GRAT Certificates

Our gratgen tool reads a DIMACS CNF formula and a DRAT certificate, and produces a GRAT certificate. Instead of the simple forward checking algorithm, it uses a multithreaded backwards checking algorithm, which is outlined below:

```
fun forward_phase:
  F := unitprop(F); if F == conflict then exit "s UNSAT"

  for item in certificate do
    case item of
      delete C => F := remove_clause(F,C)
    | add C =>
        F := F ∧ C
        F := unitprop(F);
        if F == conflict then truncate certificate; return

  exit "s ERROR Certificate did not yield a conflict"

fun backward_phase(F):
  for item in reverse(certificate) do
    case item of
      delete C => F := F ∧ C
    | add C =>
```

[2] We found that more than 90% of the execution time is spent on unit propagation.
[3] LRAT [6] uses a similar streaming optimization, called *incremental mode*.

```
        remove_clause(F,C); undo_unitprop(F,C)
        if is_marked(C) && acquire(C) then
          if not hasRAT(C,F) then exit "s ERROR Lemma doesn't have RAT"

fun main:
  F := F₀  // F₀ is formula to be certified as UNSAT
  forward_phase
  for parallel 1..N do
    backward_phase(copy(F))
  collect and write out certificate
```

The forward phase is similar to forward checking, but does not verify the lemmas. The backward phase iterates over the certificate in reverse order, undoes the effects of the items, and verifies the lemmas. However, only *marked* lemmas are actually verified. Lemmas are marked by unit propagation, if they are required to produce a conflict. This way, lemmas not required for any conflict need not be verified nor included into the enriched certificate, which can speed up certificate generation and reduce the certificate size. Moreover, we implement core-first unit propagation, which prefers marked lemmas over unmarked ones, aiming at reducing the number of newly marked lemmas. While backwards checking and core-first unit propagation are already used in drat-trim, the distinguishing feature of gratgen is its parallel backward phase: Verification of the lemmas is distributed over multiple threads. Each thread has its own copy of the clause database and watch lists. The threads only synchronize to ensure that no lemma is processed twice (each lemma has an atomic flag, and only the thread that manages to acquire it will process the lemma), and to periodically exchange information on newly marked lemmas (using a spinlock protected global data structure).

We implemented gratgen in about 3k lines of heavily documented C++ code.

2.4 Checking GRAT Certificates

We have formalized GRAT certificate checking in Isabelle/HOL [12], and used program refinement techniques [8,10] to obtain an efficient verified implementation in Standard ML, for which we proved:

```
theorem verify_unsat_impl_correct:
  <DBi ↦ₐ DB>
    verify_unsat_impl DBi prf_next F_end it prf
  <λresult. DBi ↦ₐ DB * ↑(¬isl result ⟹ formula_unsat_spec DB F_end)>
```

This Hoare triple states that if `DBi` points to an integer array holding the elements `DB`, and we run `verify_unsat_impl`, the array will be unchanged, and if the return value is no exception, the formula represented by the range `1...F_end` in the array is unsatisfiable. For a detailed discussion of this correctness statement, we refer the reader to [7, 9]. Similarly, we also defined and proved correct a `verify_sat_impl` function.

The gratchk tool contains the `verify_unsat_impl` and `verify_sat_impl` functions, a parser to read formulas into an array, and the logic to stream the

proof file. As the correctness statement does not depend on the parameters `prf_next`, `prf`, and `it`, which are used for streaming and iterating over the lemmas, the parser is the only additional component that has to be trusted.

The formalization is about 12k lines of Isabelle/HOL text, and gratchk is 4k lines of Standard ML, of which 3.5k lines are generated from the formalization by Isabelle/HOL.

2.5 Novel Optimizations

Apart from multithreading, gratgen includes two key optimizations that make it faster than drat-trim, even in single-threaded mode: First, we implement core-first unit propagation by using separate watch lists for marked and unmarked clauses. Compared to the single watch list of drat-trim, our approach reduces the number of iterations over the watch lists in the inner loop of unit propagation, while requiring some more time for marking a lemma.

Second, if we encounter a run of RAT lemmas with the same pivot literal, we only collect the candidate clauses once for the whole run. As RAT lemmas tend to occur in runs, this approach saves a significant amount of time compared to drat-trim's recollection of candidates for each RAT lemma.

3 Benchmarks

We have benchmarked GRAT with one and eight threads against drat-trim and (incremental) LRAT [6] on the 110 problems from the 2016 SAT competition main track that CryptoMiniSat could prove unsatisfiable, and on the 128 problems that silver medalist Riss6 proved unsatisfiable. Although not among the Top 3 solvers, we included CryptoMiniSat because it seems to be the only prover that produces a significant amount of RAT lemmas.

Using a timeout of 20,000 seconds (the default for the 2016 SAT competition), single-threaded GRAT verified all certificates, while drat-trim and LRAT timed out on two certificate, and segfaulted on a third one. For fairness, we exclude these from the following figures: GRAT required 44 hours, while drat-trim required 72 hours and LRAT required 93 hours. With 8 threads, GRAT ran out of memory for one certificate. For the remaining 234 certificates, the wall-clock times sum up to only 21 hours.

The certificates from CryptoMiniSAT contain many RAT lemmas, and thanks to our RAT run optimization, we are more than two times faster than drat-trim, and three times faster than LRAT. (17h/42h/51h) The certificates from Riss6 contain no RAT lemmas at all, and we are only slightly faster. (26h/30h/42h) The scatter plot in Figure 1 compares the wall-clock times for drat-trim against GRAT, differentiated by the SAT solver used to generate the certificates.

We also compare the memory consumption: In single threaded mode, gratgen needs roughly twice as much memory as drat-trim, with 8 threads, this figure increases to roughly 7 times more memory. Due to the garbage collection in Standard ML, we could not measure meaningful memory consumptions for

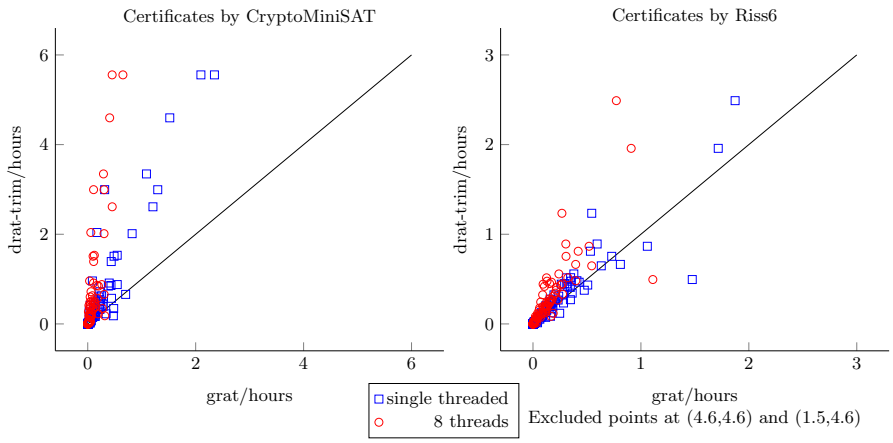

Fig. 1. Comparison of drat-trim and GRAT, ran on a server board with a 22-core XEON Broadwell CPU @2.2GHz and 128GiB RAM.

gratchk. The extra memory in single-threaded mode is mostly due to the proof being stored in memory, the extra memory in multithreaded mode is due to the duplication of data for each thread.

The certificates for the 64 satisfiable problems that CryptoMiniSat solved at the 2016 SAT competition main track [13] have a size of 229 MiB and could be verified in 40 seconds.

4 Conclusion

We have presented a formally verified (un)satisfiability certificate checker, which is faster than the unverified state-of-the-art tool. An optional multithreaded mode makes it even faster, at the cost of using more memory. Our tool utilizes a two-phase approach: The highly optimized unverified gratgen tool produces an enriched certificate, which is then checked by the verified gratchk tool.

Future Work We plan to reduce memory consumption by writing out the proof on the fly, and by sharing the clause database between threads. While the former optimization is straightforward, the latter has shown a significant decrease in performance in our initial experiments: Reordering of the literals in the clauses by moving watched literals to the front seems to have a positive effect on unit propagation, which we have not fully understood. However, when using a shared clause database, we cannot implement such a reordering, and the algorithm performs significantly more unit propagations before finding a conflict. Note that parallelization at the level of unit propagation is conjectured to be hard [5].

Acknowledgement We thank Simon Wimmer for proofreading, and the anonymous reviewers for their useful comments.

References

1. L. Cruz-Filipe, M. Heule, W. Hunt, M. Kaufmann, and P. Schneider-Kamp. Efficient certified RAT verification. In *Proc. of CADE*. Springer, 2017. To appear.
2. L. Cruz-Filipe, J. Marques-Silva, and P. Schneider-Kamp. Efficient certified resolution proof checking. In *Proc. of TACAS*, pages 118–135. Springer, 2017.
3. DRAT-trim homepage. `https://www.cs.utexas.edu/~marijn/drat-trim/`.
4. DRAT-trim issue tracker. `https://github.com/marijnheule/drat-trim/issues`.
5. Y. Hamadi and C. M. Wintersteiger. Seven challenges in parallel SAT solving. *AI Magazine*, 34(2):99–106, 2013.
6. M. Heule, W. Hunt, M. Kaufmann, and N. Wetzler. Efficient, verified checking of propositional proofs. In *Proc. of ITP*. Springer, 2017. To appear.
7. P. Lammich. Gratchk proof outline. `http://www21.in.tum.de/~lammich/grat/outline.pdf`.
8. P. Lammich. Refinement to Imperative/HOL. In *ITP*, volume 9236 of *LNCS*, pages 253–269. Springer, 2015.
9. P. Lammich. Efficient verified (UN)SAT certificate checking. In *Proc. of CADE*. Springer, 2017. To appear.
10. P. Lammich and T. Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In *Proc. of ITP*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.
11. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. of DAC*, pages 530–535. ACM, 2001.
12. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
13. SAT competition, 2016. `http://baldur.iti.kit.edu/sat-competition-2016/`.
14. N. Wetzler, M. J. H. Heule, and W. A. Hunt. Mechanical verification of sat refutations with extended resolution. In *Proc. of ITP*, pages 229–244. Springer, 2013.
15. N. Wetzler, M. J. H. Heule, and W. A. Hunt. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In *Proc. of SAT 2014*, pages 422–429. Springer, 2014.