# Efficient Certified RAT Verification*

Luís Cruz-Filipe[1], Marijn Heule[2], Warren Hunt[2],
Matt Kaufmann[2], and Peter Schneider-Kamp[1]

[1] Department of Mathematics and Computer Science
University of Southern Denmark
{lcf,petersk}@imada.sdu.dk
[2] Department of Computer Science
The University of Texas at Austin
{marijn,hunt,kaufmann}@cs.utexas.edu

**Abstract.** Clausal proofs have become a popular approach to validate the results of SAT solvers. However, validating clausal proofs in the most widely supported format (DRAT) is expensive even in highly optimized implementations. We present a new format, called LRAT, which extends the DRAT format with hints that facilitate a simple and fast validation algorithm. Checking validity of LRAT proofs can be implemented using trusted systems such as the languages supported by theorem provers. We demonstrate this by implementing two certified LRAT checkers, one in Coq and one in ACL2.

## 1 Introduction

Satisfiability (SAT) solvers are used in many applications in academia and industry, for example to check the correctness of hardware and software [7,10,11]. A bug in such a SAT solver could result in an invalid claim that some hardware or software model is correct. In order to deal with this trust issue, we believe a SAT solver should produce a proof of unsatisfiability [18]. In turn, this proof can and should be validated with a trusted checker. In this paper we will present a method and tools to do this efficiently.

Early work on proofs of unsatisfiability focused on resolution proofs [32,14]. In short, a resolution proof states for each new clause how to construct it via resolution steps. Resolution proofs are easy to validate, but difficult and costly to produce from today's SAT solvers [20]. Moreover, several state-of-the-art solvers use techniques, such as automated re-encoding [24] and symmetry breaking [12,21], that go beyond resolution and therefore cannot be expressed using resolution proofs.

An alternative method is to produce *clausal proofs* [15,28,19], that is, sequences of steps that each modify the current formula by specifying the deletion of an existing clause or the addition of a new clause. Such proofs are supported by all state-of-the-art SAT solvers [8]. The most widely supported clausal proof

* Supported by the National Science Foundation under grant CCF-1526760.

format is called DRAT [16], which is the format required by the recent SAT competitions[3]. The DRAT proof format was designed to make it as easy as possible to produce proofs, in order to make it easy for implementations to support it [30]. DRAT checkers increase the confidence in the correctness of unsatisfiability results, but there is still room for improvement, i.e., by checking the result using a highly-trusted system [2,22,27]. The only mechanically-verified checkers for DRAT [31] or RUP [14] are too slow for practical use. This holds for certified SAT solving [9,25,26] as well.

Our tool chain works as follows. When a SAT solver produces a clausal proof of unsatisfiability for a given formula, we validate this proof using a fast non-certified proof checker, which then produces an optimized proof with hints. Then, using a certified checker, we validate that the optimized proof is indeed a valid proof for the original formula. We do not need to trust whether the original proof is correct. In fact, the non-certified checker might even produce an optimized proof from an incorrect proof.

Validating clausal proofs is potentially expensive [30]. For each clause addition step in a proof of unsatisfiability, unit clause propagation (explained below) should result in a conflict when performed on the current formula, based on an assignment obtained by negating the clause to be added. Thus, we may need to propagate thousands of unit clauses to check the validity of a single clause addition step. Scanning over the formula thousands of times for a single check would be very expensive. This problem has been mitigated through the use of watch pointers. However, validating clausal proofs is often costly even with watch pointers.

In this paper we first present the new expressive proof format LRAT and afterwards show that this proof format enables the development of efficient certified proof checkers. This work builds upon previous work of some of the co-authors [13], as the LRAT format and the certified Coq checker presented here extend the GRIT format and the certified Coq checker presented there, respectively. Additionally, we implemented an efficient certified checker in the ACL2 theorem proving system, extending [31].

The LRAT format poses several restrictions on the syntax in order to make validation as fast as possible. Each clause in the proof must be suitably sorted. This allows a simple check that the clause does not contain duplicate or complementary literals. Hints are also sorted in such a way that they become unit from left to right. Finally, resolution candidates are sorted by increasing clause index; this allows scanning the formula once.

This paper is structured as follows. In Section 2 we briefly recapitulate the checking procedure for clausal proofs based on the DRAT format. The novel LRAT format is introduced in Section 3. Section 4 presents an algorithm for verifying LRAT proofs, and discusses its worst-case complexity. We demonstrate the benefits of LRAT by extracting two certified checkers for the format: one in Coq (Section 5) and one in ACL2 (Section 6). We evaluate the checkers and the potential of LRAT in Section 7. Finally, we draw some conclusions in Section 8.

[3] see `http://satcompetition.org`

## 2 Background on Clausal Proof Checking

Consider a *formula*, or set of *clauses* implicitly conjoined, where each clause is a list of *literals* (Boolean proposition letters or their negations), implicitly disjoined. Satisfiability (SAT) solvers decide the question of whether a given formula is *satisfiable*, that is, true under some assignment of *true* and *false* values to the Boolean proposition letters of the formula. A formula is *unsatisfiable* if there is no assignment under which the formula is *true*.

*Example 1.* Consider the formula below, which we will use as a running example:

$$F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_4)$$

Each step in a clausal proof is either the addition or the deletion of a clause. Each clause addition step should preserve satisfiability; this should be checkable in polynomial time. The polynomial time checking procedure is described in detail below. Clause deletion steps need not be checked, because they trivially preserve satisfiability. The main reason to include clause deletion steps in proofs is to reduce the computational and memory costs to validate proofs.

A clause with only one literal is called a unit clause. Checking whether a clause is entailed by a CNF formula is computed via Unit Clause Propagation (UCP). UCP works as follows: For each unit clause $(l)$ all literal occurrences of $\bar{l}$ are removed from the formula. Notice that this can result in new unit clauses. UCP terminates when either no literals can be removed or when it results in a conflict, i.e., all literals in a clause have been removed.

Let $C$ be a clause. $\overline{C}$ denotes the negation of a clause, which is a conjunction of all negated literals in $C$. A clause $C$ has the property Asymmetric Tautology (AT) with respect to a CNF formula $F$ iff UCP on $F \wedge (\overline{C})$ results in a conflict. The core property used in the DRAT format is Resolution Asymmetric Tautology (RAT). A clause $C$ has the RAT property with respect to a CNF formula $F$ if it has the AT property or there exists a literal $l \in C$ (the *pivot*) such that for all clauses $D$ in $F$ with $\neg l \in D$, the clause $C \vee (D \setminus \{\neg l\})$ has the property AT with respect to $F$. In this case, $C$ can be added to $F$ while preserving satisfiability.

DRAT proof checking works as follows. Let $F$ be the input formula and $P$ be the clausal proof. At each step $i$, the formula is modified. The initial state is: $F_0 = F$. At step $i > 0$, the $i^{th}$ line of $P$ is read. If the line has the prefix `d`, then the clause $C$ described on that line is removed: $F_i = F_{i-1} \setminus \{C\}$. Otherwise, if there is no prefix, then $C$ must have the RAT property with respect to formula $F_{i-1}$. This must be validated. Recall that the RAT property requires a pivot literal $l$. In the DRAT formula it is expected that the first literal in $C$ is the pivot. If the RAT property can be validated, then the clause is added to the formula: $F_i = F_{i-1} \wedge C$. If the validation fails, then the proof is invalid.

The empty clause, typically at the end of the proof, should have the AT property as it does not have a first literal.

## 3 Introducing the LRAT Format

The Linear RAT (LRAT) proof format is based on the RAT property, and it is designed to make proof checking as straightforward as possible. The purpose of LRAT proofs is to facilitate the implementation of proof validation software using highly trusted systems such as theorem provers. An LRAT proof can be produced when checking a DRAT proof with a non-certified checker (cf. the end of this section).

The most costly operation during clausal proof validation is finding the unit clauses during unit propagation. The GRIT format [13] removes this problem by requiring proofs to include hints that list all unit clauses. This makes it much easier and faster to validate proofs, because the checker no longer needs to find the unit clauses. However, the GRIT format does not allow checking of all possible clauses that can be learned by today's SAT solvers and are expressible in the DRAT format.

The LRAT format extends the GRIT format to remove this limitation, by adding support for checking the addition of clauses justified by the non-trivial case of the RAT property. For efficiency, the LRAT format requires that all clauses containing the negated pivot be specified. Furthermore, for each resolvent it has to be specified how to perform UCP as is done for AT in the GRIT approach.

While the LRAT format is semantically an extension of the GRIT format, we updated two aspects. First, the clauses from the original CNF are *not* included, as this required verification that these clauses do indeed occur in the original CNF. The advantage of working only with a subset of clauses from the original CNF can be achieved by starting with a deletion step for clauses not relevant for the proof. Second, the syntax of the deletion information has been extended to include a clause identifier. To be recognized, deletion statements are now identified with lines that start with an index followed by "d". This change makes the format stable under permutations of lines. In practice, checkers expect proof statements in ascending order, which easily can be achieved by sorting the lines numerically.

To demonstrate these two changes, we first consider an example, which does *not* require the RAT property. Figure 1 shows an original CNF, the DRUP proof obtained by a SAT solver, the GRIT version of that proof, and, finally, the equivalent LRAT proof.

To specify the addition of a clause justified by the RAT property, we extend the format used for the AT property in GRIT. The line starts with the clause identifier of the new clause followed by the 0-terminated new clause. The first literal of the new clause is required to be the pivot literal. Next, for each clause with clause identifier $i$ containing the negated pivot, we specify the (negative) integer $-i$ followed by a (possibly empty) list of (positive) clause identifiers used in UCP of the new clause with clause $i$.

For example, consider the first line of the LRAT proof in Figure 2:

CNF formula

```
p cnf 4 8
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
-1 -3 -4 0
 1  3  4 0
-1  2  4 0
 1 -2 -4 0
```

DRUP format

```
       1  2 0
d  1 -3  2 0
       1  3 0
d  1  4  3 0
          1 0
d     1  3 0
d     1  2 0
d  1 -4 -2 0
          2 0
d -1  4  2 0
d  2 -4  3 0
             0
```

GRIT format

```
 1  1  2 -3 0 0
 2 -1 -2  3 0 0
 3  2  3 -4 0 0
 4 -2 -3  4 0 0
 5 -1 -3 -4 0 0
 6  1  3  4 0 0
 7 -1  2  4 0 0
 8  1 -2 -4 0 0
 9  1  2 0       1 6 3 0
                   0 1 0
10  1  3 0       9 8 6 0
                   0 6 0
11    1 0     10 9 4 8 0
             0 10 9 8 0
12    2 0     11 7 5 3 0
                0 7 3 0
13       0 11 12 2 4 5 0
```

LRAT format

```
 9 1 2 0       1 6 3 0
 9               d 1 0
10 1 3 0       9 8 6 0
10               d 6 0
11   1 0    10 9 4 8 0
11           d 10 9 8 0
12   2 0    11 7 5 3 0
12               d 7 3 0
13     0 11 12 2 4 5 0
```

**Fig. 1.** A CNF formula and three similar proofs of unsatisfiability in the DRUP, GRIT and LRAT format, respectively. Formula clauses are shown in green, deletion information in blue, learned clauses in red, and unit propagation information in yellow. The proofs do not have clauses based on the RAT property. The spacing shown aims to improve readability, but extra spacing does not effect the meaning of a LRAT file.

```
9 1 0 -2 6 8 -5 1 8 -7 6 1 0
```

The first number, `9` expresses that the new clause will get identifier 9. The numbers in between the identifier and the first `0` express the literals in the clause. In clause of clause `9` this is only literal `1`. After the first `0` follow the hints. All hints are clause identifiers or their negations. Positive hints express that the clause becomes unit or falsified. Negative hints express that the clause is a candidate for a RAT check, i.e., it contains the complement of the pivot. In the example line, there are three such negative hints: `-2`, `-5`, and `-7`. The LRAT format prescribes that negative literals are listed in increasing order of their absolute value.

After a negative hint there may be positive hints that list the identifiers of clauses that become unit and eventually falsified. For example, assigning the literal in the new clause (`1`) and the literals in the second clause apart from the negated pivot (`2` and `-3`) to false causes the sixth clause to become unit (`4`), which in turn falsifies the eigth clause.

There are two extensions to this kind of simple RAT checking. (1) It is possible that there are no positive hints following a negative hint. In this case, the new clause and the candidate for a RAT check have two pairs of complementary literals. (2) It is also possible that some positive hints are listed before the first

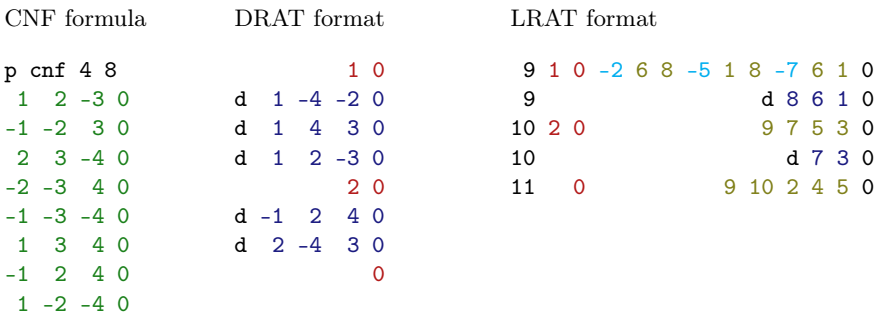


**Fig. 2.** The LRAT format with the RAT property (with original clauses in green, deletion information in blue, learned clauses in red, unit propagation information in yellow, and resolution clauses in cyan).

$$
\begin{aligned}
\langle proof \rangle &= \{\langle line \rangle\} \\
\langle line \rangle &= (\langle rat \rangle \mid \langle delete \rangle), \text{“}\backslash n\text{”} \\
\langle rat \rangle &= \langle id \rangle, \langle clause \rangle, \text{“0”}, \langle idlist \rangle, \{\langle res \rangle\}, \text{“0”} \\
\langle delete \rangle &= \langle id \rangle, \text{“}d\text{”}, \langle idlist \rangle, \text{“0”} \\
\langle res \rangle &= \langle neg \rangle, \langle idlist \rangle \\
\langle idlist \rangle &= \{\langle id \rangle\} \\
\langle id \rangle &= \langle pos \rangle \\
\langle lit \rangle &= \langle pos \rangle \mid \langle neg \rangle \\
\langle pos \rangle &= \text{“1”} \mid \text{“2”} \mid \ldots \\
\langle neg \rangle &= \text{“-”}, \langle pos \rangle \\
\langle clause \rangle &= \{\langle lit \rangle\}, \text{“0”}
\end{aligned}
$$

**Fig. 3.** EBNF grammar for the LRAT format.

negative hint. In this case, these clauses (i.e., whose identifiers are listed) become unit after assigning the literals in the new clause to false.

The full syntax of the LRAT format is given by the grammar in Figure 3, where for the sake of sanity, whitespace (tabs and spaces) is ignored. Note that syntactically, AT and RAT lines are both covered by RAT lines. AT is just the special case where there is a non-empty list of only positive hints.

Producing LRAT proofs directly from SAT solvers would add significant overhead both in runtime and memory usage, and it might require the addition of complicated code. Instead, we extended the DRAT-trim proof checker [16] to emit LRAT proofs. DRAT-trim already supported the emitting of optimized proofs in the DRAT and TraceCheck+ formats. DRAT-trim emits an LRAT proof after validation of a proof using the “`-L proof.lrat`” option.

We implemented an uncertified checker for LRAT in C that achieves runtimes comparable to the one from [13] on proofs without RAT lines.

## 4 Verifying LRAT Proofs

We now discuss how to check an LRAT proof. The algorithm we present takes as input a formula in CNF and an LRAT proof, and returns `YES` and a new CNF, or `NO`. In the affirmative case, the output CNF is satisfiable if the input CNF was satisfiable. We are thus able both to check unsatisfiability (if the formula returned contains the empty clause) and addition of clauses preserving satisfiability.

The algorithm assumes a CNF to be a finite map from a set of positive integers to clauses. We write $C_i$ for the clause with index $i$. The main step is checking individual RAT steps, which is done by Algorithm check_RAT.

Algorithm: check_RAT
Input: a CNF $\varphi = \{C_i\}_{i \in \mathcal{I}}$ and a line $\ell$ representing a RAT step.
Recall that clauses are lists of literals; we use ++ for append.

1. parse $\ell$ as $\left[j, C_j, 0, \widetilde{i^0}, \{-i^k, \widetilde{i^k}\}_{k=1}^{n}\right]$
   instantiating all variables with (vectors of) positive integers
2. set $C \leftarrow C_j$
3. for $i \in \widetilde{i^0}$
   3.1. set $C'_i \leftarrow C_i \setminus C$
   3.2. if $C'_i = \emptyset$, return `YES`
   3.3. if $|C'_i| \geq 2$, return `NO`
   3.4. set $C \leftarrow C{+}{+}\bar{C}'_i$
4. set $p \leftarrow (C)_1$
   (if $C = \emptyset$, return `NO`)
5. for $i \in \mathcal{I}$
   5.1. if $C_i$ does not contain $\bar{p}$, skip
   5.2. if $C_i$ and $C$ contain dual literals aside from $p$ and $\bar{p}$, skip
   5.3. find $j$ such that $i^j = i$ (from $\ell$)
        (return `NO` if no such $j$ exists)
   5.4. set $C' \leftarrow C{+}{+}(C_i \setminus \{\bar{p}\})$
   5.5. for $m \in \widetilde{i^j}$
        5.5.1. set $C'_m \leftarrow C_m \setminus C'$
        5.5.2. if $C'_m = \emptyset$, skip to next iteration of step 5.
        5.5.3. if $|C'_m| \geq 2$, return `NO`
        5.5.4. set $C' \leftarrow C'{+}{+}\bar{C}'_m$
   5.6. return `NO`
6. return `YES`

Steps 2 and 3 perform UCP on $\varphi \wedge \overline{C_j}$ using the clauses referred to by $i^0_1, \ldots, i^0_n$. If the empty clause is derived at some stage, then $C_j$ has the AT property w.r.t. $\varphi$. Otherwise, we store the extended clause $C$ and let $p$ be its first element (step 4). We then check that this clause has the RAT property: we go through all clauses in $\varphi$; steps 5.1 and 5.2 deal with the trivial cases, while steps 5.4 and 5.5 again perform UCP to show that $C'$ has the AT property. If

the algorithm terminates and returns `YES`, we have successfully verified that $C_j$ satisfies the RAT property with respect to $\varphi$.

Algorithm check_LRAT is now simple to define.

> Algorithm: check_LRAT
> Input: a CNF $\varphi = \{C_i\}_{i \in \mathcal{I}}$ and an LRAT proof.
>
> 1. for each line $\ell$
>  1.1. if $\ell$ can be parsed as $\langle delete \rangle$
>  then remove all clauses $C_i$ with $i \in \langle idlist \rangle$ from $\varphi$
>  1.2. if $\ell$ can be parsed as $\langle rat \rangle$, call check_RAT with $\varphi$ and $\ell$
>  1.2.1. if the result is `YES`, add $C_j$ to $\varphi$
>  1.2.2. if the result is `NO`, return `NO`
>  1.3. if $\ell$ cannot be parsed, return `NO`
> 2. return `YES` and $\varphi$

**Lemma 1 (Termination).** *Algorithm check_LRAT always terminates.*

*Proof.* Straightforward, as all cycles in both algorithms are `for` loops.

**Theorem 1 (Soundness).** *If the result of running check_LRAT on $\varphi$ and an LRAT proof is `YES` and $\varphi'$, then: (i) the proof is valid and (ii) if $\varphi$ is satisfiable, then $\varphi'$ is also satisfiable.*

We skip the proof of this theorem, as this algorithm has been directly translated to ACL2 and proved sound therein (Section 6).

We now discuss the complexity of these algorithms. We assume efficient data structures, so that e.g. finding an element in a collection can be done in time logarithmic in the number of elements in the collection. In particular, literals in clauses are ordered, and we have constant-time access to any position in a clause. The main challenge is analysing the complexity of a single RAT check.

**Lemma 2.** *Algorithm check_RAT runs in time*

$$\mathcal{O}\left(|\mathcal{I}| \cdot |\ell| \cdot (\log |\mathcal{I}| + c \cdot \log(\max(c, |\ell|)))\right) ,$$

*where $c$ is the number of literals in the longest clause in $\varphi$ and $|\ell|$ is the length of the input line.*

*Proof.* Steps 2, 4 and 6 can obviously be done in constant time, while step 1 can be done in time linear in $|\ell|$. Furthermore, the loop in step 3 is the same as that in step 5.5 (starting with $|C| \leq |C'|$), so the worst-case asymptotic complexity of the whole algorithm is that of step 5.

At the start of step 5, $|C| \leq |\ell|$: each literal in $C$ comes either from $C_j$ (which is part of $\ell$) or from one iteration of step 3, whose hint is obtained from $\ell$. Similarly, $|C'| \leq c + |\ell|$ throughout the whole cycle: its literals come either from $C_i \in \varphi$, from $C$, or from an iteration of step 5.5, whose hint is in a different part of $\ell$ than that used to build $C$.

Step 5.1 requires looking for a literal in $C_i$, which can be done in time $\mathcal{O}(\log c)$. Step 5.2 requires looking for $|C_i|$ literals in $C$, which can be done in time $\mathcal{O}(c \cdot \log(|\ell|))$. Step 5.3 requires finding an index in the data structure generated from $\ell$ in step 1, which can be done in time $\mathcal{O}(\log |\ell|)$. Steps 5.4 and 5.6 can be done in constant time.

We now analyze the loop in step 5.5, observing that it is executed at most $|\ell|$ times. The loop begins by retrieving $C_m$ from $\varphi$, which can be done in time $\mathcal{O}(\log |\mathcal{I}|)$ if we assume CNFs to be stored e.g. in a binary tree. Step 5.5.1 then removes all elements of $C'$ from $C_m$, which can be done efficiently by going through $C_m$ and checking whether each element is in $C'$; this has a global complexity of $\mathcal{O}(c \cdot \log(c + |\ell|))$. (Note that, in the successful case – the one we are interested in – the result is always the empty clause or a single literal.) All the remaining steps can be done in constant time, so the total time required by 5.5 is $\mathcal{O}(|\ell|(\log |\mathcal{I}| + c \cdot \log(c + |\ell|)))$.

Since step 5 is executed $|\mathcal{I}|$ times, the total time for the whole algorithm is thus

$$\mathcal{O}\left(|\mathcal{I}| \cdot (\log c + c \cdot \log |\ell| + \log |\ell| + |\ell|(\log |\mathcal{I}| + c \cdot \log(c + |\ell|)))\right) .$$

Since both $\log c$ and $\log |\ell|$ are bounded by $\log(c + |\ell|)$, we can replace $\log c + c \cdot \log(c + |\ell|) + \log |\ell|$ by $(c + 2) \log(c + |\ell|)$, obtaining

$$\mathcal{O}\left(|\mathcal{I}| \cdot ((c + 2) \cdot \log(c + |\ell|) + |\ell|(\log |\mathcal{I}| + c \cdot \log(c + |\ell|)))\right)$$

which we can simplify to

$$\mathcal{O}\left(|\mathcal{I}| \cdot (|\ell| \log |\mathcal{I}| + (|\ell| + 2) \cdot c \cdot \log(c + |\ell|))\right)$$

or, equivalently,

$$\mathcal{O}\left(|\mathcal{I}| \cdot |\ell| \cdot (\log |\mathcal{I}| + c \cdot \log(c + |\ell|))\right)$$

since $|\ell|$ and $|\ell| + 2$ are asymptotically equivalent. Observing that $\log(c + |\ell|) \leq \log(2(\max(c, |\ell|))) = \log(2) + \log(\max(c, |\ell|))$ yields the bound in the lemma.

**Theorem 2 (Complexity).** *The complexity of checking an LRAT proof is at most*

$$\mathcal{O}\left(n \cdot (|\mathcal{I}| + n) \cdot l \left(\log(|\mathcal{I}| + n) + k \cdot \log k\right)\right)$$

*where $n$ is the number of lines in the DRAT proof, $l$ is the length of the longest line in the proof, $\mathcal{I}$ and $c$ are as before, and $k = \max(c, l)$.*

*Proof.* The bound follows from observing that the loop in algorithm check_LRAT is executed $n$ times (in case of success); in the worst case, all steps are RAT steps, adding one clause to $\varphi$ (hence the increase in $|\mathcal{I}|$ to $|\mathcal{I}| + n$) and potentially making the size of the longest clause in $\varphi$ increase to $l$ (hence raising the multiplicative factor from $c$ to $k$ in the rightmost logarithmic term).

We make some observations. If we allow only the lengths of the proof $n$ to grow while keeping all other parameters fixed, the asymptotic complexity of

check_LRAT is $\mathcal{O}(n^2 \log n)$. Similarly, if we compare proofs of the same length but consider variations of the length of the clauses in the original CNF, the asymptotic complexity is $O(c \log c)$. In practice, we observe that algorithm check_RAT typically terminates in step 3.2; in these cases, the bound in Lemma 2 can be improved to $\mathcal{O}(|\ell| \cdot (\log |\mathcal{I}| + c \log c))$.

## 5 Checking LRAT Proofs in Coq

Our development of a verifier of LRAT proofs in Coq does not follow algorithm check_LRAT directly. This is due to the fact that we had previously developed a certified checker for GRIT proofs [13], by extracting an OCaml program from a Coq formalization, and we opted for extending this construction. In particular, the addition of clauses justified by AT (where check_RAT returns YES in step 3.2) is verified using the original checker.

The complexity of checking the RAT property in our development is better than the theoretical upper bound, because we preprocess the LRAT proof and add additional information to bypass step 5.2 when it fails. (This preprocessing amounts to checking the proof with an untrusted verifier, so the overall complexity including this step is still that of Theorem 2.) The rationale for this preprocessing is that there is a big overhead in using extracted data structures (see [23]), which means that, even if the overall complexity of the extracted checker is optimal, there are large constants that slow down the checker's performance in practice. We work with a pure extracted program, where all data structures are extracted from their Coq formalizations.[4] This means, in particular, that we do not have lists with direct access. Thus, clauses are represented as binary search trees, which allows most of the operations to have optimal complexity; the exception is the addition in steps 3.4 and 5.5.4, which takes time logarithmic in the size of the original clause, but which is dominated by other steps in the corresponding cycles.

Our experiments show that, with the optimizations enabled by preprocessing, this checker is fast enough to be used in the largest instances available.

The development of the checker in [13] is modular, with different functions that verify each type of line in a GRIT proof. We thus extended this set of functions with a function RAT_check that verifies RAT lines. This function implements a modified variant of algorithm check_RAT: the enriched proof indicates whether we should execute step 5.2 (and if so, it tells us which literal to look for). Its soundness theorem states that, if the check succeeds, then the clause given can be added to the CNF preserving satisfiability. The term c is the given CNF, while the clause $C_j$ is (pivot::cl) (so the pivot is already singled out), and L contains the remaining information in the line justifying the RAT step.

```
Theorem RAT_theorem : ∀ c pivot cl L, RAT_check c pivot cl L = true →
      ∀ V, satisfies V c →
      ∃ V, satisfies V (CNF_add (pivot::cl) c).
```

[4] With the exception of integers, which are only used as labels and therefore can be extracted to a native type without compromising soundness of the extracted code.

(For readibility, we omit type injections from the Coq listings.)

We then enrich the overall loop to include the case where the proof includes RAT lines, and reprove the correctness of the main function `refute` from [13], whose task it is to prove unsatisfiability of a given formula. Its arguments are only the CNF `c` (given as a list of pairs index/clause) and the preprocessed LRAT proof (whose type is formalized as `Oracle`).

```
Theorem refute_correct : ∀ (c:list (ad * Clause)) (O:Oracle),
  refute c O = true → unsat c.
```

By extracting `refute` we again obtain a correct-by-construction checker for proofs of unsatisfiability using the full LRAT format. If this checker returns `true` when given a particular CNF and proof, this guarantees that the CNF is indeed unsatisfiable. The universal quantification over the oracle ensures that any errors in its implementation (and in particular in the interface connecting it to the checker) do not affect the correctness of this answer.

*Satisfiability-preserving addition of clauses.* Algorithm check_LRAT is formulated not in terms of unsatisfiability, but of preservation of satisfiability – with unsatisfiability being a particular case where the empty clause is added. In order to provide this functionality, we tweaked our checker to return a pair consisting of a boolean value and a CNF. In the base case (when the input proof is empty), the checker now returns `true` (instead of `false`) together with the CNF currently stored. If the empty clause is derived at some point, the checker still returns `true` as before, but now together with a CNF containing only the empty clause. If any step fails, we return `false` and also provide the formula currently stored (which results from applying the longest initial segment of the LRAT proof that is verifiable); otherwise we proceed as before.

With these changes, we can still verify unsatisfiability as before, but we can also provide a target CNF and check that the oracle provides a correct reduction from the initial CNF to the target. Function `enrich` offers this new functionality.

```
Theorem enrich_correct : ∀ (c c':list (ad * Clause)) (O:Oracle),
  enrich c c' O = true → ICNF_reduces c c'.
```

(The predicate `ICNF_reduces` states that any valuation satisfying `c` can be used to construct a valuation satisfying `c'`.)

*Results.* After adapting the interface to be able to transform proofs in the full LRAT format into the oracle syntax defined above, we tested the extracted checker on 225 unsatisfiability proofs output by SAT solvers supporting RAT proofs. See Section 7 for further details.

We also used the possibility of adding new clauses to check the transformation proof from [17], the only SAT-related step in the original proof of the Boolean Pythagorean Triples problem that we were unable to verify in [13]. The certified LRAT checker in Coq was able to verify this proof in 8 minutes and 25 seconds, including approx. 15 seconds for checking that the formula generated by the proof coincides with the formula produced by the original SAT solver.

## 6 LRAT Checker in ACL2

In this section, in order to demonstrate the general applicability of our approach, we extended the original ACL2-based DRAT checker [29] to permit the checking of UNSAT proofs in the LRAT format. We have certified this extension using the ACL2 theorem-proving system.

We outline our formalization below using the Lisp-style ACL2 syntax, with comments to assist readers unfamiliar with Lisp syntax. Note that embedded comments begin with a "`;`" character and continue to the end of a line.

We omit the code here but note that it has been optimized for efficiency. In particular, applicative hash tables represent formulas, and are heuristically cleaned on occasion after deletion; and mutable objects [5] are used for assignments. These techniques reduce the complexity substantially. Of course, correctness of such optimizations was necessarily proved as part of the overall correctness proof. The code and top-level theorem are available from the top-level file `top.lisp` in the full proof development [1], included in the GitHub repository [6] that holds ACL2 and its libraries. Also see the `README` file in that directory. Here we focus primarily on the statement of correctness.

The top-level correctness theorem is as follows.

```
(defthm main-theorem
  (implies
   (and (formula-p formula)           ; Valid formula and
        (refutation-p proof formula)) ; Valid proof with empty clause
   (not (satisfiable formula))))      ; Imply unsatisfiable
```

The command `defthm` is an ACL2 system command that demands that the ACL2 theorem-proving system establish the validity of the claim that follows the name (in this case `main-theorem`) of the theorem to be checked.

The theorem above is expressed in terms of the three functions `formula-p`, `refutation-p`, and `satisfiable`. The first of these recognizes structures that represent sets of clauses; our particular representation uses applicative hash tables [4]. The function `refutation-p` recognizes valid proofs that yield a contradiction; thus, it calls other functions, including one that performs the necessary RAT checks. We verify an alleged proof by checking that each of its steps preserves satisfiability.

Finally, we define `satisfiable` to mean that there exists an assignment satisfying a given formula. The first definition says that the given assignment satisfies the given formula, while the second uses an existential quantifier to say that *some* assignment satisfies the given formula.

```
(defun solution-p (assignment formula)
  (and (clause-or-assignment-p assignment)
       (formula-truep formula assignment)))

(defun-sk satisfiable (formula)
  (exists assignment (solution-p assignment formula)))
```

Before our SAT proof checker can be called, an LRAT-style proof is read from a file, and during the reading process it is converted into an internal Lisp format that is used by our checker. Using the ACL2 theorem prover, we have verified the theorem `main-theorem` above, which states that our code correctly checks the validity of a proof of the empty clause.

*Results.* The ACL2 checker is able to check the validity of adding each of the 68,667 clauses in the transformation proof from [17] in less than 9 seconds. The certified checking of this LRAT proof is almost as fast as non-certified checking and conversion of the DRAT proof into the LRAT proof by DRAT-trim. This is a testament to the efficiency potential of the LRAT format in particular, and the approach taken in our work in general. At the moment of writing, the correspondence between the formula generated by the original SAT solver and by executing the proof has not been implemented yet to the ACL2 checker, but this can easily be added in a similar way as we did for the Coq checker.

## 7 Experimental Evaluation

In order to evaluate the potential of the LRAT format, we performed exensive experiments on benchmarks from the 2016 SAT competition and the 2015 SAT race. The set of instances we considered consists of the 241 instances from the main and parallel tracks that could be shown to be UNSAT within 5,000 seconds using the 2016 competition version of CryptoMiniSat v5 [3]. (CryptoMiniSat was the only solver from this competition where we were able to obtain a non-trivial number of RAT lines in most proofs.) All experiments were performed on identical nodes equipped with dual Intel Xeon E5-2680v3 running at 2.50 GHz with 64 GByte RAM on CentOS with a 3.10.0 Linux kernel.

For each of these instances, the original CNF and proof were first trimmed and optimized and then output in LRAT using drat-trim in backward checking mode. A total of 225 out the 241 instances could be successfully processed by drat-trim within 20,000 seconds. Out of the remaining 16 instances, 12 timed out, 3 resulted in a segmentation fault and 1 proof could not be verified. In total there were 381,468,814 lines in the 225 proofs totalling 250 GByte, out of which 3,260,037 were non-trivial RAT lines.

The Coq checker verified 161 out of these 225 instances within a maximum runtime of 24 hours. For the remaining 64 instances, it timed out (59), ran out of memory (1), or determined that the proof was invalid (4). The 161 verified proofs amount to a total of 88 GByte and were processed in just under 3 weeks of CPU time, or in other words at an average speed of 3 MByte per minute.

The ACL2 checker verified 212 out of the 225 instances within a maximum runtime of 6,708 seconds, typically being at least an order of magnitude faster than the Coq checker. For the remaining 13 instances, it ran out of memory (1), crashed (1), or determined that the proofs were invalid (11). The 212 verified proofs amount to a total of 205 GByte and were processed in just under 17 hours of CPU time, or in other words at an average speed of 207 MByte per minute.

The alleged LRAT proofs for the 11 instances, where verification using the ACL2 checker failed, range in size from 50 MByte to 6.4 GByte. The Coq checker either agrees with the result (7) or times out (4). We then inspected the smallest alleged proofs by hand and found that they indeed are not valid LRAT proofs.

To summarize the experiments, both certified checkers have been found to be able to verify LRAT proofs of up to several GByte within reasonable computational resources. The input data, the executables, and instructions how to rerun the experiments are available from: `http://imada.sdu.dk/~petersk/lrat/`

## 8 Conclusions

We have introduced a novel format for clausal proof checking, *Linear RAT* (LRAT), which extends the GRIT format [13] to support checking all techniques used in state-of-the-art SAT solvers. We have shown that it allows for implementing efficient certified proof checkers for UNSAT proofs with the RAT property, both using Coq and using ACL2. The ACL2 LRAT checker is almost as fast as — and in some cases even faster than — non-certified checking by DRAT-trim of the corresponding DRAT proof. This suggests that certified checking can be achieved with a reasonable overhead.

Furthermore, we have shown that our Coq checker's ability to check transformation proofs has allowed us to check the transformation proof from [17], the only SAT-related step in the original proof of the Boolean Pythagorean Triples problem that we were unable to verify in [13].

## References


1. ACL2 LRAT checker. `https://github.com/acl2/acl2/tree/master/books/projects/sat/lrat/`, Accessed: 2016.
2. The Coq proof assistant. `https://coq.inria.fr/`, Accessed: 2016.
3. Cryptominisat v5. `http://baldur.iti.kit.edu/sat-competition-2016/solvers/main/cmsat5_main2.zip`, Accessed: 2017.
4. ACL2 Community. ACL2 documentation topic: FAST-ALISTS. `http://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html?topic=ACL2____FAST-ALISTS`, Accessed: 2016.
5. ACL2 Community. ACL2 documentation topic: STOBJ. `http://www.cs.utexas.edu/users/moore/acl2/v7-2/manual/?topic=ACL2____STOBJ`, Accessed: 2016.
6. ACL2 Community. ACL2 system and libraries on GitHub. `https://github.com/acl2/acl2/tree/master/books/projects/sat/lrat/`, Accessed: 2016.
7. F. I. amd Zijiang Yang, M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2008.
8. T. Balyo, H. M. J. H., and M. Järvisalo. Sat competition 2016: Recent developments. In *AAAI 2017*, 2017.
9. J. C. Blanchette, M. Fleury, and C. Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. In *IJCAR*, pages 25–44, 2016.
10. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

11. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *CAV*, pages 436–453. Springer, 2001.
12. J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *KRÕ96*, pages 148–159. Morgan Kaufmann, 1996.
13. L. Cruz-Filipe, J. Marques-Silva, and P. Schneider-Kamp. Efficient certified resolution proof checking. In *TACAS*, LNCS, accepted for publication.
14. A. Darbari, B. Fischer, and J. Marques-Silva. Industrial-strength certified SAT solving through verified SAT proof checking. In *ICTAC*, pages 260–274, 2010.
15. E. I. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *DATE*, pages 10886–10891, 2003.
16. M. Heule. The DRAT format and DRAT-trim checker. CoRR, abs/1610.06229, 2016. Source code available from: `https://github.com/marijnheule/drat-trim`.
17. M. Heule, O. Kullmann, and V. W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *SAT 2016*, pages 228–245, 2016.
18. M. J. H. Heule and A. Biere. Proofs for satisfiability problems. In *All about Proofs, Proofs for All (APPA)*, July 2014. `http://www.easychair.org/smart-program/VSL2014/APPA-index.html`.
19. M. J. H. Heule, W. A. Hunt Jr., and N. D. Wetzler. Trimming while checking clausal proofs. In *FMCAD*, pages 181–188, 2013.
20. M. J. H. Heule, W. A. Hunt Jr., and N. D. Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Softw. Test., Verif. Reliab.*, 24(8):593–607, 2014.
21. M. J. H. Heule, W. A. Hunt Jr., and N. D. Wetzler. Expressing symmetry breaking in DRAT proofs. In *CADE*, pages 591–606, 2015.
22. M. Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Software Eng.*, 23(4):203–213, 1997.
23. P. Letouzey. Extraction in Coq: An overview. In *CiE 2008*, volume 5028 of *LNCS*, pages 359–369. Springer, 2008.
24. N. Manthey, M. J. H. Heule, and A. Biere. Automated reencoding of boolean formulas. In *Proceedings of Haifa Verification Conference 2012*, 2012.
25. F. Maric. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theor. Comput. Sci.*, 411(50):4333–4356, 2010.
26. F. Maric and P. Janicic. Formalization of abstract state transition systems for SAT. *Logical Methods in Computer Science*, 7(3), 2011.
27. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.
28. A. Van Gelder. Producing and verifying extremely large propositional refutations - have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.
29. N. Wetzler, M. J. Heule, and J. Warren A. Hunt. Mechanical verification of SAT refutations with extended resolution. In *ITP 2013*, volume 7998 of *LNCS*, pages 229–244. Springer, 2013.
30. N. Wetzler, M. J. H. Heule, and W. A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *SAT*, pages 422–429, 2014.
31. N. D. Wetzler, H. M. J. H., and W. A. Hunt Jr. Mechanical verification of SAT refutations with extended resolution. In *ITP*, pages 229–244, 2013.
32. L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, pages 10880–10885, 2003.