Scalable Trusted SAT Solving with on-the-fly LRAT Checking

Dominik Schreiber
Karlsruhe Institute of Technology, Germany

Abstract

Recent advances have enabled powerful distributed SAT solvers to emit proofs of unsatisfiability, which renders them as trustworthy as sequential solvers. However, this mode of operation is still lacking behind conventional distributed solving in terms of scalability. We argue that the core limiting factor of such approaches is the requirement of a single, persistent artifact at the end of solving that is then checked independently (and sequentially). As an alternative, we propose a bottleneck-free setup which exploits recent advancements in producing and processing LRAT information to immediately check all solvers' reasoning *on-the-fly* during solving. In terms of clause sharing, the guarantee of a derived clause's soundness is transferred from the sending to the receiving side via cryptographic signatures. Experiments with up to 2432 cores (32 nodes) indicate that our approach reduces the running time overhead incurred by proof checking by an order of magnitude, down to a median overhead of $\leq 42\%$ over non trusted solving.

2012 ACM Subject Classification Hardware → Theorem proving and SAT solving; Theory of computation → Automated reasoning; Computing methodologies → Distributed algorithms

Keywords and phrases SAT solving, distributed algorithms, proofs

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500).

Acknowledgements The author thanks Marijn J. H. Heule, who inspired this work; Florian Pollitt et al. [34] who enabled this work with their enhancements to CaDiCaL; Peter Sanders for helpful discussions and valuable remarks; and Patrick Hegemann for double-checking cryptographic details.

1 Introduction

The Propositional Satisfiability (SAT) problem, i.e., to satisfy a given Boolean expression or to report its unsatisfiability, is an essential building block at the core of automated reasoning, symbolic AI, and formal verification [16]. Due to its high practical relevance, increasingly efficient SAT solving approaches have emerged over the last decades, prompting a plethora of applications [9, 32, 36, 37, 42, 45, 48] to use SAT solvers as efficient blackbox engines. Today, researchers and industrial users increasingly aim to exploit distributed environments [8, 11, 18] to push the frontier of problems that are feasible to solve.

An important topic in SAT research is the reliability and trustworthiness of solvers [5, 7, 31, 33]. In particular, sequential solvers have been able for many years to output *proofs of unsatisfiability*—witnesses for a formula's unsatisfiability which can be verified by independent and even formally verified *proof checkers* [28]. For the longest time, the best performing parallel and distributed solvers were missing this crucial feature (*cf.* [24]). Only recently, Michaelson et al. proposed a feasible approach to produce proofs with parallel and distributed clause-sharing SAT solving [33]. The technology which enables this approach is a clausal proof format called LRAT [12], where each clause derivation has a unique ID and explicitly references the prior clauses required to check the derivation. This information allows to feasibly reconstruct a single proof from many LRAT-producing SAT solvers which run in parallel and periodically exchange clauses in an *all-to-all* fashion [33].

Even after these advances, the scalability of trustworthy parallel and distributed solving

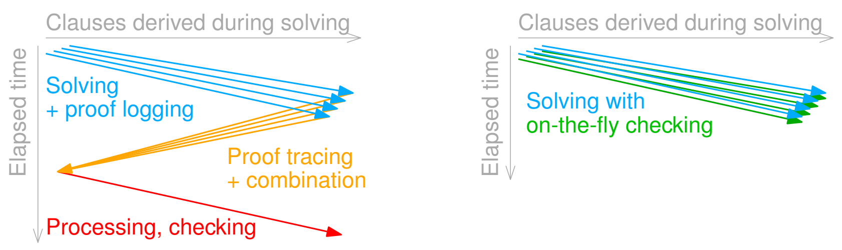


Figure 1 Schematic overview on a prior approach [33] (left) and our new approach (right) to trusted parallel solving. Each line, very roughly speaking, corresponds to an execution thread.

remains lacking. In Michaelson et al.'s procedure, all relevant proof information is funneled into a single process and then output *sequentially*. As such, the production and checking of a combined proof is invariably throttled by the (I/O) bandwidth at the final process. Likewise, to our knowledge all LRAT checkers so far are sequential. Note that proof size increases both with solving time and, to a lesser degree, the number of solvers [33]. Maintaining the active clauses in huge proofs may further slow down the proof checker or even cause main memory shortage at some point. For all these reasons put together, we argue that the production and checking of proofs currently poses a distinct bottleneck for trusted distributed solving.

To address this conceptual problem, we believe that it is sensible to slightly relax the task at hand. Even if trusting a solver's result is crucial, a witness for this result in the shape of a persistent artifact may often be expendable. Dropping this requirement, we are able to replace Michaelson et al.'s three-stage procedure with a *single-stage* procedure that finishes whenever solving finishes (see Fig. 1). We build upon a suggestion by Marijn Heule to check proofs *during solving* via inter-process communication and raise this idea to a scalable level: Each solver thread now forwards each derivation to a checker process, which checks the derivation at once. To transfer the guarantee of a clause's soundness from one checker to another, each checker outputs a cryptographic *signature* for each clause it checked. The receiving solver's checker then validates the signature by recomputing it. Given a secret key shared among all checkers, we show this procedure to be as trustworthy as *post-mortem* proof checking: a solver bug is substantially less likely to create an unsound clause with a valid signature than a proof checker is to "hallucinate" unsatisfiability due to a hardware-side memory error. Our critical code is a small, isolated, and simple C99 codebase whose formal verification we believe to be a distinct possibility in the near future.

We implemented our approach using the LRAT-producing solver CaDiCaL [34] and the distributed system MallobSat [39, 40]. For a fair comparison, we applied a number of updates to MallobSat with proof production [33]. Experiments on up to 2432 cores (32 nodes) indicate that our approach incurs a median slowdown of 40% (at 76 cores) to 42% (at 2432 cores) over conventional, non trusted solving. In comparison, the median slowdown of producing and checking a combined proof surpasses 330% at 76 cores and 750% at 1216 cores. As such, our approach's overhead is smaller by an order of magnitude and can enable trusted solving in cases where explicit proof production is infeasible. We anticipate a number of use cases for this technology, ranging from debugging parallel solvers over distributed solving with error-prone or insecure communication to formally verified scalable SAT solving.

In Section 2, we introduce preliminaries and prior work on the subject. We describe our approach in Section 3 and present an experimental evaluation in Section 4. In Section 5, we discuss the possible ramifications of our contributions. Section 6 concludes our work.

2 Preliminaries

We discuss some relevant preliminaries to the work at hand with a focus on proofs of unsatisfiability in sequential, parallel, and distributed SAT solving.

2.1 SAT Solving

We consider formulas in *conjunctive normal form* (CNF), i.e., of the form $F = \bigwedge_{i=1}^{k} \bigvee_{j=1}^{n_i} l_{ij}$, where each of the k disjunctions is called a *clause of length* n_i and each *literal* l_{ij} is a Boolean variable or its negation. The *Boolean satisfiability* (SAT) problem is to assign a value to all Boolean variables in F such that F evaluates to `true` or to recognize that this is impossible. F is called *satisfiable* if such an assignment exists and *unsatisfiable* otherwise. We say that a clause c is *sound* (w.r.t. F) if and only if c is a logical consequence of F, i.e., $F \wedge \neg c$ is unsatisfiable. As a special case, F is unsatisfiable if and only if clause $c = \emptyset$ is sound w.r.t. F.

Today's most popular and most efficient sequential SAT solvers build upon the *Conflict-Driven Clause Learning* (CDCL) paradigm: The solver searches the space of partial variable assignments and derives a *redundant conflict clause* whenever it encounters a conflict with the current assignment [13]. These redundant clauses are crucial for pruning search space and for eventually deriving the *empty clause*, i.e., the contradiction in an unsatisfiable formula.

The currently most scalable parallel and distributed SAT solvers are *clause-sharing portfolio solvers*, where many (mostly CDCL) solver threads run in parallel on the original formula and periodically exchange promising learned conflict clauses [2]. We recently confirmed that clause sharing is in fact the main driver of the scalability of the state-of-the-art distributed solver MallobSat and can ensure good performance even if running identical, i.e., non diversified solver threads [40]. There are other parallelization approaches to SAT with explicit search space splitting [23, 44] which we do not cover here.

We refer to the author's dissertation [38] for a more detailed introduction to sequential, parallel, and distributed SAT solving in the context of the work at hand.

2.2 Proofs of Unsatisfiability

If a formula F is satisfiable, any found satisfying assignment can be verified in linear time by evaluating F under the assigned values. Since today's solvers commonly output such a satisfying assignment when reporting satisfiability, we do not consider their trustworthiness to be a notable issue for satisfiable problems. By contrast, if F is unsatisfiable and a witness for the claimed unsatisfiability is desired, the solver must produce a *proof of unsatisfiability*. Such a proof contains the solver's chain of reasoning which lead to the empty clause. Proofs can be checked by independent and sometimes formally verified *proof checkers* [28] and can be useful for reasons beyond trust, e.g., for Minimally Unsatisfiable Subset (MUS) extraction [3] or for analyzing the efficiency [43] or scalability [26] of solving approaches.

In this work we focus on the LRAT (*Linear Reverse Asymmetric Tautology*) clausal proof format [12]. Simply put, LRAT proof information output by a solver reflects the solver's changes to its clause set: If a new clause is derived, a *clause addition* (or *derivation*) is appended to the proof, and if a clause is discarded, a corresponding *deletion statement* is appended to the proof. The LRAT format requires each derivation of a clause c to label the new clause with a unique ID, $id(c)$, and to explicitly include the required *dependencies* for this derivation, often called *hints*, in the form of a sequence D_c of clause IDs. A deletion statement features a sequence of IDs which refer to the clauses to delete.

LRAT proof checkers traverse the proof information at hand in a single linear pass. A checker first receives all original problem clauses F and initializes its own clause set $C := F$. It then successively applies the proof's individual clause additions and deletions to C. Clause deletions are strictly speaking not required but are important in practice since they significantly reduce the memory footprint of proof checking [21]. Each clause addition is checked using the *LRAT criterion* or, in many cases, a simpler special case named the *LRUP criterion*. While the former corresponds to the most powerful proof format known [27], the latter currently covers most solvers' reasoning, including CaDiCaL's (which we use for our study) [34]. For a derivation of clause c with dependencies D_c, LRUP requires that the clause set $\{\{\bar{l}\} : l \in c\} \cup \{c' : id(c') \in D_c,\ c' \in C\}$ results in an efficiently computable conflict: Asserting the negated literals of c, the sequence of clauses referenced by D_c must break down into unit clauses and, finally, the empty clause. As such, the checker confirms that $C \wedge \neg c$ results in unsatisfiability. Therefore, c is sound w.r.t. C. The derivation of the empty clause $\tilde{c} := \emptyset$ poses a special case where the unsatisfiability of C, and hence F, is testified.

Few parallel solvers support proof production. One of them is Gimsatul [17], an integrated shared-memory solver written from scratch. Its architecture allows for outputting a single consistent DRAT proof from all solver threads. Unlike LRAT, the earlier DRAT format [20] does not feature any explicit dependency information, which renders DRAT proofs easier to produce but substantially more expensive to check [12]. Gimsatul's approach is limited to shared memory, and combining multiple DRAT proofs into a single proof has, so far, largely resulted in proofs that are infeasible to check [24, 33].

Michaelson et al. [33] recently proposed a more general approach to proof production that is viable in distributed environments. It features the following stages:

1. **Solving**: All solver threads write their LRAT proof information to individual *partial proof* files. Clause IDs are assigned in a globally unique manner: All clause IDs produced by a certain solver are pairwise congruent modulo the total number of solvers.
2. **Combination**: As soon as a solver finds the empty clause, the solving procedure is *rewound*. All solver threads read their respective partial proofs in reverse and trace all transitive dependencies of the found empty clause. Each clause sharing operation is reversed by redistributing the IDs of required *remote* clauses back to their origin. All clause derivations identified as required are funneled into a single process, which writes the combined and dependency-ordered LRAT proof information to a single file. Note that this information cannot be streamed to a checker directly since it is still reversed.
3. **Checking**: The combined proof is read *from back to front*, which amounts to the correct chronological order, and is validated by a sequential LRAT checker.

In its original form, this proof production approach featured additional pre– and postprocessing stages which were required due to limitations of the used solver backend [33]. Since then, Pollitt et al. published a version of CaDiCaL with full LRAT support [34] which renders these steps obsolete. We outline according updates in some more detail in Section 4.2.1.

3 Trusted Solving Approach

We now present our approach to trusted parallel and distributed solving.

3.1 Overview

Our approach exploits a suggestion by Marijn Heule for the setting of sequential SAT solving: Given a SAT solver that "natively" produces LRAT information [34], its proof output can be redirected to a concurrently running LRAT checking process which checks the solver's

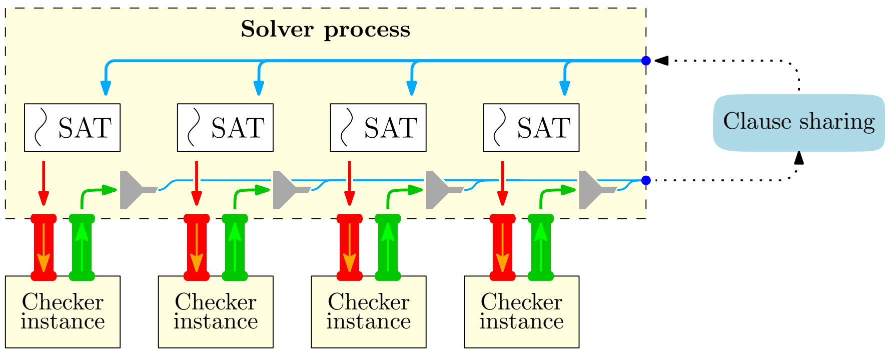


Figure 2 Information flow in a solver process with our approach. Each SAT solver thread emits LRAT proof information, which is streamed to the solver thread's corresponding checker instance (red pipe downwards). A checker produces *signatures* for checked clauses (green pipe upwards). Some of the *signed clauses* are exported and exchanged across processes. Incoming shared clauses are forwarded to the corresponding solvers. Successfully imported clauses, together with their signatures, are as well streamed to the respective checker.

reasoning *on-the-fly*.[1] This can be achieved with inter-process communication (e.g., UNIX pipes) and hence without writing proof information to disk. Note that this method is not necessarily viable for the earlier and still more widely used DRAT proof format [20] since DRAT checking is substantially more expensive and therefore likely to stall solving.

Our distributed solver employs p *solver threads* distributed across m *processes*, with $t = p/m$ solver threads per process. Each solver thread produces a stream of LRAT proof information, assigning globally unique clause IDs just like in Michaelson et al.'s approach (Section 2.2). We suggest to perform the on-the-fly checking outlined above for each solver thread, running one associated *checker instance* (i.e., LRAT checking process) each.

An important observation for our approach is that a sound clause is sound *globally*: it can safely be used regardless of its particular dependencies or a checker's internal state. As such, once a derived clause has been checked by some checker instance, other checker instances can add the clause without checking, i.e., as if it were an *axiom* or an original problem clause. The clause can then be referenced in derivations just like locally produced clauses.

Fig. 2 illustrates the resulting solving and checking setup. All produced clause derivations are first redirected to the solver's associated checker instance. Clauses can only be exported and shared after they have been checked. (This is ensured not only by our proposed architecture but also by a *signing* mechanism which we explain in detail further below.)

By checking each clause *before* exporting it to other solvers, it appears obvious that each clause used in the distributed solving procedure has been checked at some point, and one may argue that this is sufficient to render distributed solving trustworthy. However, we do not want to rely on the correctness of distributed clause sharing and its underlying buffering and communication mechanisms. For instance, a single flipped bit in a clause sharing buffer can result in an allegedly sound clause which in fact induces a wrong result. Perhaps a more realistic scenario, a programming error which (under rare circumstances) causes reading of a buffer beyond its limit can result in "garbage clauses". As an example, the distributed solver MallobSat features multithreading, multi-processing, inter-process communication with UNIX pipes and shared memory, and distributed asynchronous message passing. All of these

[1] `https://github.com/marijnheule/coch-demo`

mechanisms are involved in clause sharing in some way. Programming such systems can be challenging, and formally verifying distributed programs is laborious and only covers selected aspects of the used technology stack [30, 47]. Even if an application is provably correct, implementations of the Message Passing Interface (MPI) are known to suffer from bugs [14]. For these reasons put together, we suggest to not make *any* assumptions on the behavior nor the correctness of a distributed solver and its underlying communication mechanisms.

In order to transfer the guarantee of a clause's soundness from one checker instance to another without relying on correct sharing, we have each checker module emit a *signature* $\mathcal{S}(c)$ for a checked produced clause c. Just like its LRAT clause ID [33], this signature is considered part of the clause during sharing. At the receiving side, the checker module validates the incoming clause's signature upon its import. Under the guarantee that only trusted modules are able to compute clause signatures, we show that this method grants full confidence in the soundness of all clauses leaving a solver-checker unit. We use the same signing mechanism to ensure that all solvers indeed operate on the intended formula.

3.2 Interface

Our solving procedure features three kinds of trusted modules: one *parser* at one particular process; p *checkers* $I_1, \ldots, I_p$ (t per process, i.e., one per solver thread); and, optionally, a *confirmer* at some process. We assume that our distributed solving procedure has a *private input* to these $p+2$ modules, namely a 128-bit key K which will be used to compute signatures.[2] K is handed to our trusted modules directly and must be inaccessible otherwise.

First, a **parser** takes a path to the input formula F. It parses F, computes the signature $\mathcal{S}(F)$ for F, and returns the pair $(F, \mathcal{S}(F))$, which can then be distributed to all processes.

Next, we describe the interface of each **checker** I, shown in Fig. 3. Note that this interface generalizes the LRAT proof format: `load` corresponds to specifying the formula, `produce` corresponds to adding clause derivations, and `delete` corresponds to deleting clauses.

The first call a checker expects is `init`, where the caller commits to the formula's signature $\mathcal{S}(F)$ in advance. I then receives F via `load`. Our interface supports multiple subsequent calls to `load`, which allows reading large formulas in chunks. Loading F must be concluded with `end_load`, at which point I recomputes the signature on the loaded literals to support that the loaded formula is indeed the one parsed before. The remaining methods are used during solving to process LRAT proof information. At each call to `produce`, the flag `share` indicates whether the clause c in question is intended to be shared. If this flag is `true`, the call returns signature $\mathcal{S}(c)$ on success. Each *incoming* clause c, before being referenced in subsequent derivations, must be introduced via `import` together with $\mathcal{S}(c)$, which the receiving checker validates by recomputing it. Deletion of clauses (`delete`) works exactly as in usual LRAT proofs. Finally, if a solver thread returns from solving and reports unsatisfiability, its associated checker instance can be queried via `validate_unsat` to confirm that the empty clause has indeed been found and validated through `load`/`endload`, `produce`, or `import`. On success, a special signature $\hat{s}$ based on $\mathcal{S}(F)$ is output, certifying the unsatisfiability of F.

All methods which allow for a failure state return a truth value indicating whether the respective operation was successful. A checker instance I which returned `false` at any point from any call will never return `true` from subsequent calls, in particular to `validate_unsat`.

Lastly, the **confirmer** can be invoked after solving to validate an alleged unsatisfiability

[2] We do not consider the problem of secure key distribution (e.g., [49]). This problem heavily depends on choosing a specific attacker model, which is not the focus of our work.

```
Requires UNINITIALIZED - Ensures INITIALIZING
init(sig: Signature) → void
Requires INITIALIZING
load(formula: ClauseSet) → void
Requires INITIALIZING - Ensures VALID or INVALID according to output
end_load() → bool
Requires VALID - Ensures VALID or INVALID according to output
produce(id: ID, lits: Clause, hints: IDList, share: bool) → (bool, Signature?)
Requires VALID - Ensures VALID or INVALID according to output
import(id: ID, lits: Clause, sig: Signature) → bool
Requires VALID - Ensures VALID or INVALID according to output
delete(ids: IDList) → bool
Requires VALID - Ensures VALID or INVALID according to output
validate_unsat() → (bool, Signature?)
Terminates checker module from any state
terminate() → void
```

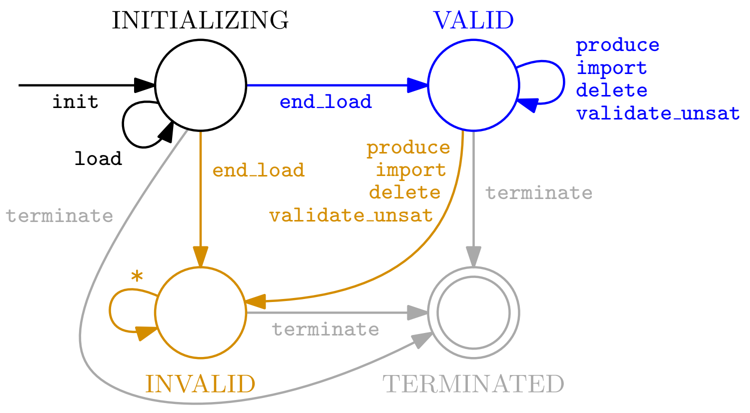


Figure 3 Checker interface with basic contracts (top) and corresponding state machine (bottom). Some directives can cause a VALID or INVALID state based on the operation's success. Transitions which violate a contract, e.g., calling `load` from VALID, all lead to INVALID and are omitted.

signature $\hat{s}$. It takes $\hat{s}$ and the path to F, parses F and computes $\mathcal{S}(F)$, internally re-computes $\hat{s}$ based on $\mathcal{S}(F)$, and outputs whether it found $\hat{s}$ to be sound. We consider this step of confirmation a necessity only if the output obtained from a parallel solver cannot be trusted.

3.3 Signatures

We now describe how we define the signature function $\mathcal{S}$ in such a way that only our trusted modules can compute it—effectively ensuring that only checked clauses can be imported.

Our signature function $\mathcal{S}(\cdot) := \mathcal{S}_K(\cdot)$ is parametrized with secret key K and evaluated in three different contexts: on a formula F, on a single clause c, or to certify unsatisfiability ("$\perp$"). We define $\mathcal{S}_K$ based on cryptographic *Message Authentification Codes* (MACs) [19], where a sender and a receiver uphold a message's authenticity by computing a signature based on a shared secret key. Specifically, we use SipHash [1], a fast and popular *keyed pseudo-random function* (PRF) which only uses addition, rotation, and XOR (*"ARX"*). For highest confidence, we use the SipHash variant which produces 128-bit signatures.

Let $H_K : \{0,1\}^* \to \{0,1\}^{128}$ be the SipHash-2-4-128 function with key K and let "||" denote concatenation of data. We define $\mathcal{S}_K$ for the three above contexts as follows:

$$\mathcal{S}_K(F) := H_K(\,F\,), \quad \mathcal{S}_K(c) := H_K(\,id(c)\,||\,c\,||\,\mathcal{S}_K(F)), \quad \mathcal{S}_K(\bot) := H_K(\,20\,||\,\mathcal{S}_K(F)).$$

Defining $\mathcal{S}_K(c)$ relative to $\mathcal{S}_K(F)$ is optional hardening to ensure that interacting checkers indeed operate on the same key and the same formula. We define $\mathcal{S}_K(\bot)$ rather arbitrarily to sign the customary return code "20" for unsatisfiability. Adding $S_K(F)$ to an input does not weaken the MAC—since $H_K(x)$ is a PRF, $H_K(x\,||\,\alpha)$ is also a PRF for any α that does not leak information on K.[3] We also ensure that all inputs are unambiguous and that our definitions have disjoint inputs: F is encoded as 4-byte integers ending with a 2-byte zero, $20\,||\,\mathcal{S}_K(F)$ is $1+16=17$ bytes long, and clause signatures are based on $8+4|c|+16$ bytes.

3.4 Correctness

We now establish the correctness of our approach given idealized signature guarantees.

▶ **Theorem 1.** *Consider our trusted solving setup for formula F. Assume that each string s which constitutes a valid signature for object x with key k indeed originates from computing $s \leftarrow \mathcal{S}_k(x)$ within our trusted code. If the solving setup outputs $\mathcal{S}_K(\bot)$, then F is unsatisfiable.*

Proof. We assume the above prerequisites. First, since only checkers can output $\mathcal{S}_K(\bot)$, there is a checker $\hat{I}$ which answered to a query `validate_unsat` with $\mathcal{S}_K(\bot)$. This implies that $\hat{I}$ received the empty clause $\hat{c} := \emptyset$ (a) via `load`/`end_load`, (b) via `produce_clause`, or (c) via `import_clause`.[4] Case (a) directly implies the unsatisfiability of F: `end_load` must have been called after all `load` calls but before `validate_unsat`, and `end_load` checks via $\mathcal{S}(F)$ that all loaded clauses, and $\hat{c}$ in particular, belong to F. In case (b), $\hat{I}$ checked $\hat{c}$, and in case (c), another checker I' signed $\hat{c}$ and therefore checked $\hat{c}$. In both cases (b) and (c), $\hat{c}$ was checked, which renders $\hat{c}$ sound if the clauses $D_{\hat{c}}$ in the derivation of $\hat{c}$ are sound. For each clause in $D_{\hat{c}}$ that is *not* an original problem clause, we recurse on the argument we just made for $\hat{c}$ itself. Since each derived clause c relies on $|D_c| > 0$ prior clauses and is added to a checker only *after* its checking, the dependencies across clauses must form a DAG whose source nodes are clauses added without checking nor dependencies, i.e., via `load`. As such, the derivation of $\hat{c}$ is sound if all clauses added via `load` by any contributing checker are original clauses of F. Since any contribution by a checker requires a prior call to `end_load` (see case (a)), which validates the loaded formula, all contributing checkers indeed operate on the intended F. The derivation of $\hat{c}$ is thus sound and F is in fact unsatisfiable. ◀

For the reverse direction, our approach can *preserve* a sequential solver's completeness:

▶ **Theorem 2.** *Consider an error-free execution of the described trusted solving setup for formula F where one solver thread is complete. If F is unsatisfiable, then $\mathcal{S}_K(\bot)$ is output.*

Proof. Consider the checker instance $\hat{I}$ of a complete sequential solver S. $\hat{I}$ functionally subsumes a plain LRAT checker which mirrors the reasoning of S. Assuming sound proof logging, $\hat{I}$ will thus receive and validate the empty clause $\hat{c}$ which S eventually derives. S then terminates with unsatisfiability, which our setup confirms by querying `validate_unsat` in $\hat{I}$. The added capabilities of our checker over plain LRAT checking cannot obstruct any of

[3] By contradiction: Any strategy distinguishing $g(x) := H_K(x\,||\,\alpha)$ from true randomness can be modified into a distinguisher for $f(x) := H_K(x)$ just by replacing each call $g(x)$ with a call $f(x\,||\,\alpha)$.
[4] The empty clause is usually not shared since its derivation renders any further sharing obsolete.

these operations. In particular, error-free clause imports can only *accelerate* the progress towards unsatisfiability. As such, $\hat{I}$ will indeed output $\mathcal{S}_K(\bot)$. ◀

Note that any call to `import` with an *incorrect* signature functionally disables the checker, which makes it impossible to export further clauses or report unsatisfiability from the respective solver. Our underlying design decision is that we are interested in every error and hence interrupt solving globally even if the solving procedure is in principle recoverable. In other settings, perhaps when faced with error-prone communication or exascale computing, it may be sensible to let checkers continue normally after rejecting a particular import.

3.5 Confidence

Let us now discuss the conditions causing the above assumptions to break in such a way that our trusted code reports unsatisfiability for a fixed satisfiable formula F. We still assume that our $p+2$ trusted processes are sound, uncompromised, and the only actors knowing K. Under these assumptions, we can discern the following *attack vectors*:

1. An unsatisfiability certificate $\mathcal{S}_K(\bot)$ is obtained for F (although F is satisfiable).
2. A checker imports a clause-signature pair (c, s) where c is unsound w.r.t. F but $\mathcal{S}_K(c) = s$, which can lead to incorrect results and hence enable attack 1 (in particular if $c = \emptyset$).
3. A formula $F' \neq F$ results in a collision $\mathcal{S}_K(F) = \mathcal{S}_K(F')$, which allows to re-use signed outputs related to F' for F. This enables attacks 1 (if F' is unsatisfiable) and 2.
4. The secret key K is recovered, which trivially enables (in particular) attack 1.

All of the outlined attacks involve either to obtain K or to otherwise *forge* a pair $(o, \mathcal{S}_K(o))$ for an object o chosen by the attacker. (For attack 3, finding the desired *second preimage* F' for $\mathcal{S}_K(F)$ implies such a forgery for $o := F'$.) MAC schemes such as SipHash claim protection against both forgery and key recovery. Specifically, if an attacker is able to guess and test 2^x 128-bit SipHash signatures for an object o of their choosing, they can find a valid signature for o with probability 2^{-128+x}. Likewise, if an attacker can guess and test 2^x values of K, they succeed at obtaining K with probability 2^{-128+x} [1]. This presumably holds even if the attacker can query signatures for objects $o' \neq o$ at will. Assuming that a single invalid signature aborts the solving procedure and that a different key K is used for each solving attempt, we have $x = 1$ which yields a success probability of around 10^{-38}. As such, successful forgery or key recovery can be ruled out for all practical purposes.[5]

We argue that the deliberate attacks outlined above cannot be outdone by inadvertent bugs or errors during distributed solving. In particular, data corruption which happens to result in a forged signature must be by pure chance (2^{-128}) as long as K is not accessed. Accidental reading of K, in turn, is ruled out since K is available only within the confined address spaces of our trusted modules. For practical purposes, if no protection against malicious intent is required, we consider the use of one fixed K for all solving attempts to be sufficient for highest confidence in the produced results.

3.6 The Road To Verified Distributed Solving

We now briefly discuss how our work relates to potential efforts towards formally verified and, in particular, provably correct parallel and distributed SAT solving.

[5] Schroeder et al. [41] estimate that around 10^4 to 10^5 DRAM errors occur per 10^9 device hours, translating to roughly 10^{-8} memory errors per second. Even if only a byte out of a terabyte (10^{-12}) in a device's main memory was critical to a checker's correctness, a memory error leading to an incorrect result would be a quintillion times more likely ($10^{-8} \cdot 10^{-12} = 10^{-20}$) than 10^{-38}.

As of yet, neither our on-the-fly checking approach nor its particular implementation are formally verified. We do consider such an undertaking a distinct possibility for future work. In particular, we believe that formal verification efforts for prior LRAT checkers [22, 46] are extensible to the additional interface methods in our trusted modules—under the assumption that our signing scheme guarantees perfect authenticity—with modest effort. Such prior works used verified toolchains like ACL2 or CakeML to produce verified code down to the machine instruction level [22, 46]. Alternatively, we may pursue direct verification of our critical code (see Section 4.1), combined with the use of a verified compiler [29].

Verified SAT solving usually encompasses the aspects of *correctness* (separable into the correctness of SAT and UNSAT results) and *termination* [5]. Verifying the correctness of our approach would equate to obtaining a verified distributed SAT solver in terms of UNSAT correctness. This can easily be extended to SAT correctness by checking the found satisfying assignment in the corresponding checker instance.[6] We consider verification in terms of termination to be of less practical interest for parallel solvers. That being said, verified termination could be achieved by running an isolated verified SAT solver, e.g., IsaSAT [5], next to the portfolio and terminating globally when that solver exits.

4 Evaluation

We now present the evaluation of our approach, beginning with a discussion of our implementation, then outlining our experimental setup and the changes we made to Michaelson et al.'s proof production approach, and finally presenting results. Our experimental data and all our code can be found via `https://github.com/domschrei/mallob-impcheck-data`.

4.1 Implementation

Our implementation bases on MallobSat [39, 40], the state of the art in distributed SAT solving, with LRAT-producing CaDiCaL [34] as a solver backend. We connected CaDiCaL's LRAT proof tracer interface, where each derivation and deletion arrives, to an interface that connects to the solver thread's checker sub-process. The solver thread itself writes each emitted LRAT directive to a single-producer, single-consumer ring buffer R and is then free to continue solving. A dedicated thread polls directives from R and forwards them to the sub-process; another thread reads results from the sub-process and reacts accordingly, e.g., by exporting a derived clause with the returned signature. As such, a solver thread never needs to wait for its checker sub-process except if it produces proof information faster than can be processed (in which case the solver thread blocks waiting for free space in R). No unsound derivation nor UNSAT result can ever leave a solver-checker unit; both cases require a valid signature and thus an explicit confirmation from the checker.

Our trusted modules are written in pure C99, which is well supported for future efforts on verification [4] and verified compilation [29]. We kept this trusted codebase isolated, small, and simple. We only use few standard library features such as memory allocation and file I/O. In terms of program instructions, our inter-process communication via named pipes is indistinguishable from linear, plain file I/O and thus does not allow for adversary inputs corrupting the program. Our implementation amounts to around 1000 effective lines of code (ELOC). In comparison, we counted around 40k ELOC in CaDiCaL, 60k ELOC in

[6] We did in fact implement such a mechanism into our checker module (see Section 4.1), with the caveat that original problem clauses can no longer be deleted during solving since they are required for checking. Our experimental evaluation does not yet include this feature.

Mallob [35], and over 300k ELOC in Open MPI (`ompi` and `opal`). As such, our approach may reduce the code critical for a distributed solver's correctness by two orders of magnitude.

We tested our code with small random manipulations during a solving procedure (e.g., inserting a superfluous dependency in a derivation or tampering with a clause literal during clause sharing), confirming that an appropriate error is output and that solving aborts.

4.2 Experimental Setup

We run all experiments on a high-performance cluster called **HoreKa** located in Karlsruhe, Germany. Each used compute node has access to 256 GB of main memory and features two Intel Xeon Platinum 8368 sockets with 38 cores each. Each core consists of two hardware threads, leading to a total of 152 hardware threads per compute node. Nodes can communicate via an InfiniBand 4X HDR interconnect.[7] We made sure that high-bandwidth output, such as proofs, is written to local disks (960 GB NVMe SSD) rather than a network file system.

We consider two solver systems in their latest version: the shared-memory solver Gimsatul [17] and the distributed solver MallobSat. Since we are unsure how well Gimsatul handles 76 cores across two sockets, we run this solver in two modes: on all 76 cores of a node, and on one socket (38 cores) only. Regarding MallobSat, we use 1–32 compute nodes (76–2432 cores) at once, thus covering the range from moderate parallelism up to a massively parallel scale. We spawn two processes per node (one per socket), run up to 38 solvers per process, and leave the remaining hardware threads to other concurrent tasks such as LRAT checking and clause sharing. This is in line with MallobSat's deployment in most earlier works [35, 39, 40], the only difference being that hardware threads not occupied with SAT solving now need to perform more work, which may in turn slow down solver threads.

Regarding MallobSat, we focus on a CaDiCaL portfolio, which proved to perform competitively in recent experiments [33, 40]. We refer to our approach as M-ImpChk ("**M**allobSat with **im**mediate **m**assively **p**arallel **p**ropositional **p**roof **ch**ec**k**ing"), to MallobSat with explicit proof production as M-Proof, and to MallobSat without any proof processing as M-nt ("**n**on **t**rusted"). Our version and configuration of MallobSat is similar to the one described recently [40] apart from the updated CaDiCaL backend (Section 4.1). Since M-ImpChk requires more RAM due to concurrent checking, we halved M-ImpChk's memory threshold per solver, which implies that the largest instances (hundreds of megabytes) are solved with fewer solvers than in M-nt or M-Proof.

We use the 400 benchmark instances from SAT Competition 2023. Note that these benchmarks contain some crafted instances which are designed to be difficult to solve with usual reasoning [6, 10]. MallobSat can solve some of them when using Lingeling (without proofs; see [25]). Since we use CaDiCaL, we ignore the according offset in performance and leave trusted solving with advanced reasoning for future work. We allow up to 300 s of wallclock time for solving and up to 1500 s of wallclock time each for combining and for checking a proof. For M-Proof we consider three timings: the *solving time* (ST), which is the time span from program start to a solver reporting a result; the "*time until proof*" (TuP), which is the time span from program start to the presence of a single proof artifact that is checkable in a single linear pass; and the "*time until validation*" (TuV), which corresponds to the TuP plus the time required to check the proof. For M-ImpChk, there is no TuP; its TuV corresponds to its ST, since we stop the ST at the point where an unsatisfiability result is *confirmed* by a checker instance. We assess an approach in terms of its ST / TuP / TuV

[7] `https://www.nhr.kit.edu/userdocs/horeka/hardware/`

overhead, which is the ratio between its ST / TuP / TuV and the ST of M-NT, minus one.

4.2.1 Updates to Proof Production

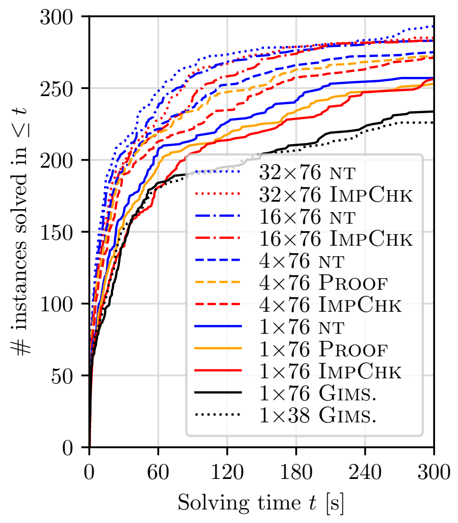
We now outline the changes we made to M-PROOF in order to make its running times more competitive and to render proof production and checking more feasible in our experiments. We may publish these changes separately and in more detail at a later point.

The updated CADICAL backend allowed us to remove a previously required sequential preprocessing step which exhaustively performed unit propagation on the input. We correspondingly do not need to produce and prepend a proof for this preprocessing. Moreover, the original setup featured postprocessing where the inverted combined proof is un-inverted and syntactically transformed to feature a compact domain of IDs. This step was a necessity because of relatively poor tool support for the kind of LRAT proofs M-PROOF emits (albeit perfectly valid in principle). Specifically, `lrat-check` from the `drat-trim` toolbox is not able to gracefully handle large gaps between subsequent clause IDs. The more recent checker `lrat-trim` [34] operates on 32-bit IDs and is hence not suitable either. In our setup, we use a standalone, fast LRAT checker crafted from MALLOBSAT code which can operate directly on the compressed and inverted proof. As such, we measure the TuP up to the point where the combined, compressed, and reversed proof has been written. The TuV then adds a single linear read of the proof file,[8] thus minimizing I/O operations.

4.3 Results

We first compare the solving times (ST) of all considered approaches. Fig. 4 provides an overview. Note again that this does *not* include the time required by M-PROOF to combine

[8] Our reverse file parser uses buffering to reverse-read roughly as fast as usual forward reading.



		all		satisf.		unsat.	
	# nodes	#	PAR	#	PAR	#	PAR
M-NT	1	257	241.5	120	248.7	137	234.3
	4	275	211.5	124	233.9	151	189.1
	16	283	196.2	126	229.2	157	163.3
	32	293	184.2	128	222.6	165	145.7
M-PROOF	1	253	250.0	119	254.1	134	245.9
	4	272	216.7	122	239.1	150	194.3
	16	282	197.4	126	228.5	156	166.4
M-IMPCHK	1	257	251.5	122	250.2	135	252.7
	4	271	222.3	124	235.0	147	209.6
	16	283	201.0	128	225.8	155	176.3
	32	285	195.3	129	221.5	156	169.1
GIMSATUL	1	234	279.1	122	246.7	112	311.6
	1/2	226	286.3	120	250.5	106	322.1

Figure 4 Left: Pure solving times of M-NT, M-PROOF (*excluding proof combination and checking*), M-IMPCHK, and GIMSATUL (*excl. proof checking*), on 1–32 nodes. Colors delineate the approaches, line styles indicate the scale of solving. "16×76 PROOF" is omitted to reduce clutter; it is closely aligned with "16×76 NT". Right: Solved instances ("#") and PAR-2 scores ("PAR") of each run.

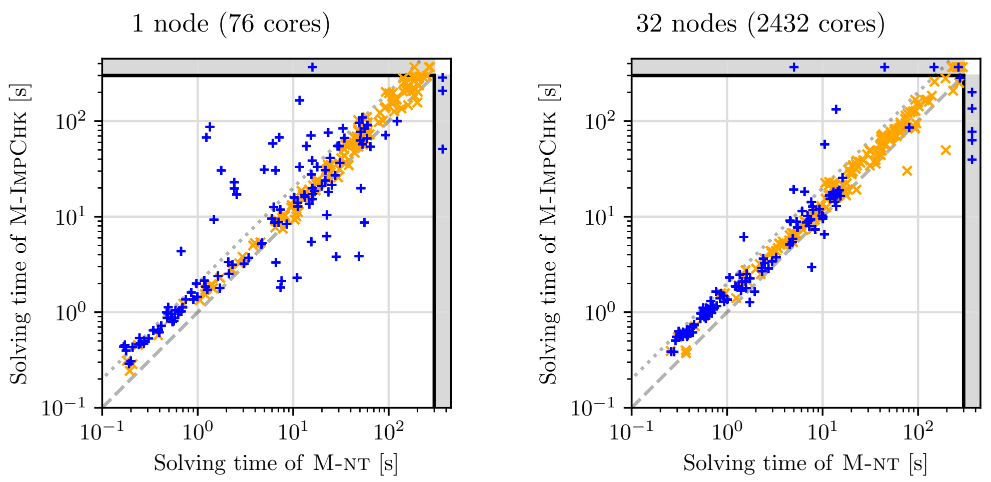


Figure 5 Solving times of M-NT vs. M-ImpChk on one (left) and 32 nodes (right). Points on the central diagonal denote instances solved equally fast; the dotted diagonal shows where M-ImpChk takes twice as long. Blue "+" denote satisfiable, orange "×" unsatisfiable instances.

and check proofs nor the DRAT proof checking time for Gimsatul. All approaches consistently show improved performance when increasing the computational resources. Gimsatul did not reach the performance of MallobSat. After our updates, M-Proof now performs very similarly to M-NT. The overhead of M-ImpChk, which we analyze below, is more noticeable. Still, M-ImpChk at 16 (4) nodes is able to outperform all approaches at 4 (1) nodes, and M-ImpChk at 32 nodes is on par with M-NT at 16 nodes.

Fig. 5 provides a detailed comparison of M-NT vs. M-ImpChk solving times at the smallest and largest scale considered. The incurred overhead is consistent, stable, and does not correlate with M-NT's solving times (Pearson's r: -0.11 at 1 node, -0.16 at 32 nodes).

Fig. 6 (leftmost plot) shows the per-instance overhead of M-ImpChk. At all scales, the observed overhead is below 70% (i.e., slowdown 1.7) for three out of four instances and at most 42% for the median instance. Specifically, the median overhead ranges from 37.1% (at four nodes) to 42.1% (at 32 nodes), with the geometric mean ranging from 37.7% (at 16 nodes) to 45.6% (at one node). In addition, we averaged the checker processes' CPU utilization during a solving attempt to analyze how much time the checkers spend on actual processing vs. waiting for the next directive. We arrived at a median utilization from 11.2% at one node to 12.4% at 32 nodes (geometric mean 10.4%–11.0%), which indicates that our checkers are mostly idle[9] and, in turn, have no difficulties with processing the produced proof information and computing signatures on-the-fly. The increased communication volume (with each clause now carrying 192 bits of metadata) appears to be unproblematic as well, which is unsurprising considering that MallobSat does not even remotely exhaust the communication bandwidth of HPC interconnects [40]. All of these observations support that our approach to trusted solving is scalable and bottleneck-free.

Fig. 6 (right) shows the overhead incurred by M-Proof. The median ST overhead is below 11% at all scales, which indicates that producing and writing LRAT proof information is inexpensive in and of itself. The TuP overhead, by contrast, is substantial and increases with the scale of solving; even at a single node (median 104%), it surpasses the TuV overhead

[9] The low CPU ratio is not a consequence of over-subscribed hardware threads. All non-solver, non-checker threads use little CPU time and thus leave plenty of CPU time for the checkers.

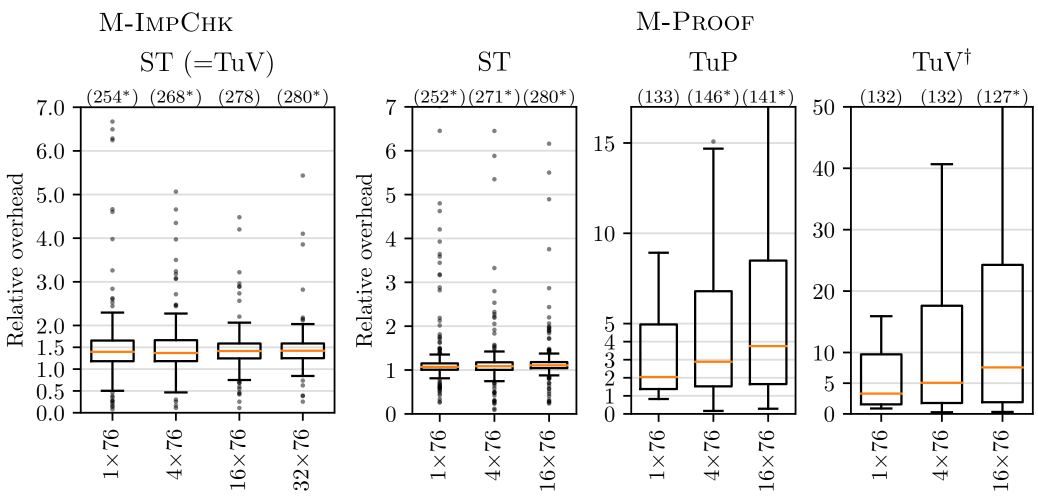


Figure 6 ST overhead (= TuV overhead) of M-ImpChk; and ST, TuP, and TuV overhead (the latter two for unsatisfiable instances only) of M-Proof. Numbers at the top show the number of considered data points. *Some data surpasses the displayed interval and has been cut off. †TuV overheads at 4 and 16 nodes are not real measurements but estimated based on the checker's proof traversal time at one node scaled by the obtained proof size at 4 and 16 nodes respectively.

of our approach at *any* scale. Going from 4 to 16 nodes, the growing TuP overhead (median 196% to 275%) even causes a decreasing number of successfully produced proofs (146 to 141).

Across the 127 instances where M-Proof produced a proof at all three tested scales, the mean proof size increases from 7.33 GiB at one node to 10.73 GiB (+46%) at four nodes and finally to 14.46 GiB (another +35%) at 16 nodes. As such, it became increasingly challenging to store and/or check all produced proofs, which exceed 4.5 TiB for the 16-node run alone. Therefore, for all runs beyond a single node, we only retained the size (in bytes) of a proof and then deleted it without checking. To still gain an impression on the TuV at 4 and 16 nodes, we extrapolated estimates based on the assumption that, for a fixed instance, a proof's checking time grows linear in the proof's file size. We estimated an instance's TuV at 4 (16) nodes by its TuP plus its one-node proof checking time, scaling the time needed to traverse the proof by the obtained proof size at 4 (16) nodes. This extrapolation only covers instances where already the single-node run produced and checked a proof, and the estimates can exceed our actual proof checking time limit of 1500 s. The estimated median TuV overhead increases steeply with the scale of solving—from 233% at one node to 416% at four nodes and to 657% at 16 nodes. Overall, our experiments and analysis have fully confirmed our concerns with respect to the scalability of M-Proof.

The complete DRAT proofs output by Gimsatul average 2.36 GiB for the 38-core variant and 3.10 GiB for the 76-core variant. Out of 97 38-core proofs which `drat-trim` deemed complete and sound, only 65 proofs were checked successfully, at a median overhead of 475% (geometric mean 629%), while 32 checking attempts timed out. This confirms that checking DRAT proofs from parallel solvers can at times border on infeasible [17, 24].

Lastly, we tracked MallobSat's peak global RAM usage (measured once a second). For the median instance at 32 nodes, M-ImpChk increased the RAM usage over M-nt by around 60%. Note that we reduced M-ImpChk's per-solver memory threshold (Section 4.2), which however was only triggered in four cases. By contrast, M-Proof incurs virtually no RAM overhead (median $< 3\%$ at all scales) since all proof information is written to disk and processed with external-memory data structures [38, Sect. 5.5].

5 Discussion

In the following, we discuss the merits and the limitations of the proposed approach.

We believe that our approach can be applied to many other clause-sharing solvers. Its requirements are that each solver thread outputs LRAT information and that each clause is shared with its LRAT ID and its signature. Unlike M-Proof, whose proof combination bases on periodic all-to-all clause sharing, our checking can be used with any clause sharing, including exchanges along rings [50] or communication graphs [15]. From a pragmatic point of view, we release our trusted modules (Section 4.1) together with introductory examples and documentation on how developers can integrate them into a parallel solver.

Our approach does not rely on I/O speed nor large amounts of disk space. While not yet implemented, it is also entirely possible to perform on-the-fly checking while additionally writing partial proofs to later serve as witnesses. Moreover, on-the-fly checking easily allows for *malleable* setups, where solvers may be added or removed during solving. This is useful for scheduling and solving many SAT instances at once [35] and for reacting to main memory shortages [40]. In addition, we anticipate our approach to be useful for solver development. Pollitt et al. found on-the-fly checking to be "*dramatically reducing the implementation effort (particularly for debugging)*" [34], which is now possible at a distributed level.

A limitation of our approach in its current form is its relatively high RAM usage. While the asymptotic memory usage of solving remains the same, an increase by a constant factor is to be expected since we now maintain *two* clause databases per solver thread—one within the solver and one in the LRAT checker. A possible measure to make RAM usage less of an issue would be to let the solver threads within a process share data structures, as in Gimsatul [17]. Similarly, our t checker sub-processes per SAT process could be combined into a *single* checker process. This process may need to be multi-threaded in order to achieve the required throughput. Still, memory usage could be reduced significantly by storing equivalent clauses only once while reference-counting them for safe deletion (cf. [24]).

All in all, our impression is that, in a few years, on-the-fly proof checking may be sufficiently advanced and widely supported that the International SAT Competition is in a position to require verified results across all principal tracks (sequential, parallel, cloud). We would consider this a significant advancement in terms of reliable and trustworthy SAT solving which can be used even for critical matters in a carefree manner.

6 Conclusion

Motivated by the insufficient scalability of producing proofs in distributed systems, we have presented a novel approach to trusted distributed clause-sharing solving where LRAT information is checked on-the-fly. The critical code is small, simple, and kept separate from solver code, and the solver's clause sharing does not need to be trusted. We confirmed our approach to be bottleneck-free and to scale drastically better than prior trusted approaches with explicit proof combination. To the best of our knowledge, this is the first instance of a trustworthy parallel solver whose running times are dominated by solving, not checking.

Regarding future work, we aim to formally verify our approach in order to obtain a verified parallel SAT solver (see Section 3.6). In addition, we are interested in adding LRAT support to further solvers, in particular Gimsatul and Kissat, to make our approach more broadly applicable, boost its performance, and reduce its main memory requirements. Lastly, we intend to add full LRAT (rather than LRUP) checking to our approach, which may help with integrating solvers with advanced reasoning into parallel and distributed solvers.

References

1 Jean-Philippe Aumasson and Daniel J Bernstein. SipHash: a fast short-input PRF. In *International Conference on Cryptology in India*, pages 489–508. Springer, 2012. doi:10.1007/978-3-642-34931-7_28.

2 Tomáš Balyo and Carsten Sinz. Parallel satisfiability. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*. Springer, 2018. doi:10.1007/978-3-319-63516-3_1.

3 Anton Belov, Marijn JH Heule, and Joao Marques-Silva. MUS extraction using clausal proofs. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 48–57. Springer, 2014. doi:978-3-319-09284-3_5.

4 Dirk Beyer. Automatic verification of C and Java programs: SV-COMP 2019. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 133–155. Springer, 2019. doi:10.1007/978-3-030-17502-3_9.

5 Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. *Journal of automated reasoning*, 61:333–365, 2018. doi:10.1007/s10817-018-9455-7.

6 Bart Bogaerts, Jakob Nordström, Andy Oertel, and Cagrı Uluç Yıldırımoglu. Crafted benchmark formulas requiring symmetry breaking and/or parity reasoning. In *SAT Competition*, page 67, 2023.

7 Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 44–57. Springer, 2010. doi:10.1007/978-3-642-14186-7_6.

8 Mark Alexander Burgess, Charles Gretton, Josh Milthorpe, Luke Croak, Thomas Willingham, and Alwen Tiu. Dagster: Parallel structured search with case studies. In *Pacific Rim Int. Conf. Artificial Intelligence*, pages 75–89. Springer, 2022. doi:10.1007/978-3-031-20862-1_6.

9 Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19:7–34, 2001. doi:10.1023/A:1011276507260.

10 Cayden R. Codel, Joseph E. Reeves, and Randal E. Bryant. Pigeon hole and mutilated chessboard with mixed constraint encodings and symmetry-breaking. In *SAT Competition*, page 72, 2023.

11 Byron Cook. Automated reasoning's scientific frontiers. https://www.amazon.science/blog/automated-reasonings-scientific-frontiers, 2021. Amazon Science.

12 Luis Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Proc. CADE*, pages 220–236, 2017. doi:10.1007/978-3-319-63046-5_14.

13 Adnan Darwiche and Knot Pipatsrisawat. Complete algorithms. In *Handbook of Satisfiability*, pages 101–132. IOS Press, 2021. doi:10.3233/faia200987.

14 Daniel DeFreez, Antara Bhowmick, Ignacio Laguna, and Cindy Rubio-González. Detecting and reproducing error-code propagation bugs in MPI implementations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–201, 2020. doi:10.1145/3332466.3374515.

15 Thorsten Ehlers, Dirk Nowotka, and Philipp Sieweck. Communication in massively-parallel SAT solving. In *Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, pages 709–716. IEEE, 2014. doi:10.1109/ictai.2014.111.

16 Johannes K. Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. The silent (r)evolution of SAT. *Comm. ACM*, 66(6):64–72, 2023. doi:10.1145/3560469.

17 Mathias Fleury and Armin Biere. Scalable proof producing multi-threaded SAT solving with Gimsatul through sharing instead of copying clauses. In *Pragmatics of SAT*, 2022.

18 Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artificial Intelligence*, 301:103572, 2021. doi:10.1016/j.artint.2021.103572.

19 Edgar N Gilbert, F Jessie MacWilliams, and Neil JA Sloane. Codes which detect deception. *Bell System Technical Journal*, 53(3):405–424, 1974. `doi:10.1002/j.1538-7305.1974.tb02751.x`.

20 Marijn J. H. Heule. The DRAT format and DRAT-trim checker. *CoRR*, abs/1610.06229, 2016. `arXiv:1610.06229`.

21 Marijn J. H. Heule, Warren Hunt, and Nathan Wetzler. Trimming while checking clausal proofs. In *Proc. FMCAD*, pages 181–188. IEEE, 2013. `doi:10.1109/fmcad.2013.6679408`.

22 Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In *Proc. ITP*, pages 269–284, 2017. `doi:10.1007/978-3-319-66107-0_18`.

23 Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Haifa Verification Conference*, pages 50–65. Springer, 2011. `doi:10.1007/978-3-642-34188-5_8`.

24 Marijn J. H. Heule, Norbert Manthey, and Tobias Philipp. Validating unsatisfiability results of clause sharing parallel SAT solvers. In *Pragmatics of SAT*, pages 12–25, 2014. `doi:10.29007/6vwg`.

25 Alexey Ignatiev, António Morgado, and João Marques-Silva. On tackling the limits of resolution in SAT solving. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 164–183. Springer, 2017. `doi:10.1007/978-3-319-66263-3_11`.

26 George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In *AAAI Conference on Artificial Intelligence*, volume 27, pages 481–488, 2013. `doi:10.1609/aaai.v27i1.8660`.

27 Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn JH Heule. Extended resolution simulates DRAT. In *International Joint Conference on Automated Reasoning*, pages 516–531. Springer, 2018. `doi:10.1007/978-3-319-94205-6_34`.

28 Peter Lammich. Efficient verified (UN)SAT certificate checking. *Journal of Automated Reasoning*, 64(3):513–532, 2020. `doi:10.1007/s10817-019-09525-z`.

29 Xavier Leroy. *The CompCert C verified compiler: Documentation and user's manual.* PhD thesis, Inria, 2023.

30 Ziqing Luo, Manchun Zheng, and Stephen F Siegel. Verification of MPI programs using CIVL. In *Proceedings of the 24th European MPI Users' Group Meeting*, pages 1–11, 2017. `doi:10.1145/3127024.3127032`.

31 Norbert Manthey, Tobias Philipp, and Christoph Wernhard. Soundness of inprocessing in clause sharing SAT solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 22–39. Springer, 2013. `doi:10.1007/978-3-642-39071-5_4`.

32 João P. Marques-Silva and Karem A. Sakallah. Boolean satisfiability in electronic design automation. In *Proc. DAC*, pages 675–680, 2000. `doi:10.1145/337292.337611`.

33 Dawn Michaelson, Dominik Schreiber, Marijn J. H. Heule, Benjamin Kiesl-Reiter, and Michael W. Whalen. Unsatisfiability proofs for distributed clause-sharing sat solvers. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 348–366. Springer, 2023. `doi:10.1007/978-3-031-30823-9_18`.

34 Florian Pollitt, Mathias Fleury, and Armin Biere. Faster lrat checking than solving with CaDiCaL. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. `doi:10.4230/LIPIcs.SAT.2023.21`.

35 Peter Sanders and Dominik Schreiber. Decentralized online scheduling of malleable NP-hard jobs. In *Proc. Euro-Par*, pages 119–135. Springer, 2022. `doi:10.1007/978-3-031-12597-3_8`.

36 André Schidler and Stefan Szeider. SAT-based decision tree learning for large data sets. In *AAAI Conference on Artificial Intelligence*, volume 35, pages 3904–3912, 2021. `doi:10.1609/aaai.v35i5.16509`.

37 Dominik Schreiber. Lilotane: A lifted SAT-based approach to hierarchical planning. *Journal of Artificial Intelligence Research (JAIR)*, 70:1117–1181, 2021. `doi:10.1613/jair.1.12520`.

38 Dominik Schreiber. *Scalable SAT Solving and its Application.* PhD thesis, Karlsruhe Institute of Technology, 2023. `doi:10.5445/IR/1000165224`.

39 Dominik Schreiber and Peter Sanders. Scalable SAT solving in the cloud. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 518–534. Springer, 2021. `doi:10.1007/978-3-030-80223-3_35`.

40 Dominik Schreiber and Peter Sanders. MallobSat: Scalable SAT solving by clause sharing. Submitted to *Journal of Artificial Intelligence Research* (JAIR), 2024. URL: `https://dominikschreiber.de/papers/2024-jair-mallobsat-pre.pdf`.

41 Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):193–204, 2009. `doi:10.1145/2492101.1555372`.

42 JOM Silva and Karem A. Sakallah. Robust search algorithms for test pattern generation. In *Proc. IEEE Int. Symp. Fault Tolerant Computing*, pages 152–161. IEEE, 1997. `doi:10.1109/ftcs.1997.614088`.

43 Laurent Simon. Post mortem analysis of SAT solver proofs. In *Pragmatics of SAT*, pages 26–40, 2014.

44 Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. PaSAT – parallel SAT-checking with lemma exchange: Implementation and applications. *Electronic Notes in Discrete Mathematics*, 9:205–216, 2001. `doi:10.1016/s1571-0653(04)00323-3`.

45 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 244–257. Springer, 2009. `doi:10.1007/978-3-642-02777-2_24`.

46 Yong Kiam Tan, Marijn J. H. Heule, and Magnus Myreen. Verified LRAT and LPR proof checking with cake_lpr. In *SAT Competition*, page 89, 2023.

47 Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M Kirby, and Rajeev Thakur. Formal verification of practical MPI programs. *ACM Sigplan Notices*, 44(4):261–270, 2009. `doi:10.1145/1594835.1504214`.

48 Sean Weaver and Marijn J. H. Heule. Constructing minimal perfect hash functions using SAT technology. In *AAAI Conference on Artificial Intelligence*, volume 34, pages 1668–1675, 2020. `doi:10.1609/aaai.v34i02.5529`.

49 Lihao Xu and Cheng Huang. Computation-efficient multicast key distribution. *IEEE Transactions on Parallel and Distributed Systems*, 19(5):577–587, 2008. `doi:10.1109/TPDS.2007.70759`.

50 Xindi Zhang, Zhihan Chen, and Shaowei Cai. PRS: A new parallel/distributed framework for SAT. In *SAT Competition*, pages 39–40, 2023.