

# Towards Practical Privacy-Preserving SAT Solving

Gefei Tan\*   Wenhao Zhang<sup>†</sup>   Timos Antonopoulos<sup>‡</sup>  
Ruzica Piskac<sup>§</sup>   Xiao Wang<sup>¶</sup>   Ning Luo<sup>||</sup>

## Abstract

Privacy-preserving Boolean satisfiability (SAT) solvers allow multiple distrustful parties to solve the conjunction of their private formulas without revealing their inputs. Prior work on privacy-preserving SAT solvers, i.e., ppSAT (USENIX Security 2022), fails to solve formulas of practical size and complexity because it supports only the most basic SAT-solving algorithm. We bring privacy-preserving SAT solving closer to practicality by introducing ppCDCL. Through carefully orchestrated oblivious data structures and solver architecture, our new solver enables conflict-driven clause learning (CDCL) and efficient propagation, the two most important features of modern plaintext SAT solvers. Evaluation results show that ppCDCL outperforms ppSAT in both capability and efficiency. It solves significantly more instances: 98% vs. 65% on the Haplotype benchmarks and 84% vs. 28% on the larger, more diverse SATLIB benchmarks. Furthermore, it solves 36% of SATLIB instances within 1,000 seconds compared to only 4% for ppSAT.

## 1 Introduction

Boolean satisfiability (SAT) solving [FM09] is a cornerstone of modern computer science, with applications in hardware and software verification [MSS00, CKL04, Kro09], model checking [BCCZ99], bioinformatics [LMS06a, LMS06b, MCP<sup>+</sup>06], automated planning [KS99, Rin21], and many more. Given a Boolean formula encoded in conjunctive normal form (CNF), a SAT solver determines whether there exists an assignment that makes the formula true. Despite SAT’s NP-completeness [Coo23], modern SAT solvers routinely solve large industrial instances with millions of variables [SE05, BFF<sup>+</sup>24, MSLM21, AFT11].

The expressiveness of CNF allows many complex problems to be encoded as SAT [BN21]. In many practical scenarios, however, the resulting formula contains sensitive constraints distributed across mutually distrustful parties. For example, consider logistics planning, where companies sharing transportation or warehouse capacity must jointly determine whether a feasible schedule exists. Planning problems have a natural SAT encoding [Rin21]: Boolean variables represent actions at each time step, and clauses enforce preconditions, effects, and resource constraints. Each party’s clauses may reveal proprietary information such as demand forecasts, capacity limits, delivery deadlines, or customer locations. Similar scenarios arise in cross-organizational security-policy and configuration analysis, where organizations may wish to determine whether their private access-control or configuration constraints are jointly consistent without disclosing the

---

\*Northwestern University, [gefeit@u.northwestern.edu](mailto:gefeit@u.northwestern.edu)

<sup>†</sup>Northwestern University, [wenhao.zhang@northwestern.edu](mailto:wenhao.zhang@northwestern.edu)

<sup>‡</sup>Yale University, [timos.antonopoulos@yale.edu](mailto:timos.antonopoulos@yale.edu)

<sup>§</sup>Yale University, [ruzica.piskac@yale.edu](mailto:ruzica.piskac@yale.edu)

<sup>¶</sup>Northwestern University, [wangxiao1254@gmail.com](mailto:wangxiao1254@gmail.com)

<sup>||</sup>University of Illinois Urbana-Champaign, [n127@illinois.edu](mailto:n127@illinois.edu)

rules themselves. Using an existing plaintext SAT solver in either setting would require the parties to expose their private constraints.

A privacy-preserving SAT solver allows parties to jointly determine whether the conjunction of their private formulas is satisfiable, and to recover a satisfying assignment when one exists, without revealing their individual constraints beyond the output. Recently, ppSAT [LJA<sup>+</sup>22] introduced the first two-party private SAT solver. ppSAT implements an oblivious variant of the classical Davis–Putnam–Logemann–Loveland (DPLL) algorithm [DP60, DLL62] inside secure two-party computation (2PC). DPLL is a backtracking search that repeatedly picks a variable, assigns a truth value, propagates the consequences of that assignment, and chronologically backtracks upon conflict. This structure is attractive in privacy-preserving settings because its regular, repetitive control flow is relatively simple to make oblivious.

However, ppSAT remains far from practical on standard SAT-solving benchmarks. For example, on several very small benchmarks from the popular SATLIB library [HS00], extrapolating ppSAT’s per-step cost predicts that it would fail to solve even after  $10^{12}$  s, while modern plaintext SAT solvers solve the same instances in less than 0.01 s. This gap cannot be attributed solely to the overhead of 2PC. Rather, it reflects a fundamental limitation of the underlying plaintext algorithm used by ppSAT: plain DPLL lacks two features that are central to the effectiveness of modern SAT solvers, namely *conflict-driven clause learning* (CDCL) and *lazy propagation data structures*. Solvers equipped with these features are often called CDCL solvers [MSLM21, MMZ<sup>+</sup>01, MSS99, SE05, BFF<sup>+</sup>24], and most high-performance complete SAT solvers used in practice are CDCL-based.

## 1.1 Privacy-Efficiency Trade-Offs in CDCL

The performance gap above suggests that private SAT solvers must incorporate CDCL to make progress toward practical relevance. The challenge is that the same mechanisms that make CDCL effective in plaintext are highly data-dependent. A plaintext solver is fast precisely because it exploits the structure of the input formula and the current search state, which is what a private solver must hide.

**Conflict analysis and clause learning.** A central ingredient of CDCL is conflict analysis followed by clause learning. When a partial assignment falsifies a clause, the solver analyzes why the conflict occurred, derives a *learned clause* that prevents the same bad search pattern from recurring, and then backtracks non-chronologically to an earlier decision level where the learned clause becomes useful. Over time, learned clauses prune large regions of the search space, which is a major reason CDCL solvers dramatically outperform plain DPLL in practice.

At the same time, conflict analysis is inherently *instance- and state-dependent*. To analyze a conflict, the solver traverses an *implication graph* whose structure depends on the input formula, the branching decisions made so far, and the sequence of assignments. Consequently, the locations of conflicts, the clauses and variables visited during analysis, the size of the learned clause, and the backtrack destination all depend on private data. Making this process oblivious in 2PC is therefore non-trivial: it requires hiding not only the learned clause itself, but also the formula-dependent execution pattern that produced it.

**Efficient propagation via lazy data structures.** An equally important reason for the success of modern SAT solvers is efficient unit propagation. Unit propagation [DLL62, ZM88] repeatedly simplifies the formula by identifying clauses that force new literal assignments and by detecting conflicts. Because this procedure is invoked after every new assignment, it typically dominates solver runtime.

Modern solvers accelerate propagation using *lazy* data structures such as two-watched literals (2WL) [ZS00, MMZ<sup>+</sup>01]. In 2WL, each clause watches two literals, and each literal has a watchlist containing the clauses currently watching it. The solver visits a clause only when one of its watched literals becomes false;

all other assignments can be ignored without affecting completeness. Empirically, this strategy avoids most clause scans on most assignments and eliminates the need to update the clause database during backtracking. As a result, propagation touches only a small fraction of clauses per assignment and becomes highly efficient in practice.

Making lazy propagation oblivious without destroying its benefit is fundamentally difficult. Its efficiency comes precisely from selective, data-dependent memory accesses: each assignment touches only the clauses that are affected, and the identities of those clauses depend on both the hidden formula and the current hidden partial assignment. In 2PC, these access patterns must themselves be hidden. Thus, the mechanism that makes propagation efficient in plaintext is exactly what makes it expensive to realize obliviously.

## 1.2 Our Contributions

This work introduces ppCDCL, an efficient privacy-preserving CDCL solver with contributions in both formal methods and cryptography:

- **A CDCL-based SAT solver in 2PC.** We build ppCDCL, a privacy-preserving SAT solver that supports advanced SAT-solving features such as clause learning and lazy propagation. Evaluation shows that ppCDCL significantly outperforms the state-of-the-art private solver ppSAT: on the SATLIB benchmark suite, ppCDCL solves **84.3%** of instances versus **27.5%** for ppSAT. On benchmark families bf, flat200, and logistics, where ppSAT solves **0** instances, ppCDCL solves **100%** of bf and logistics, and **96%** of flat200.
- **Oblivious clause learning with deletion.** We propose a clause-learning scheme for privacy-preserving CDCL that bounds the amount of conflict-analysis work and discards learned clauses that exceed a public size budget. This avoids the worst-case padding overhead that would otherwise dominate runtime, while retaining the short learned clauses that are most likely to be useful for future propagation. Empirically, our scheme improves efficiency, yielding up to a  $283\times$  speedup over a baseline that implements clause learning naively.
- **Optimized oblivious data structures for SAT solving.** We optimize and tailor oblivious data structures for the data-dependent operations central to CDCL, including watchlists for lazy propagation and clause databases for learned-clause management. Where our structures use ORAM, we further specialize each implementation to the access pattern of its corresponding solver component.

## 2 A Motivating Example

We use the following formula to illustrate (1) why CDCL mechanisms such as clause learning and two-watched literals (2WL) are highly effective in plaintext, (2) why the same mechanisms are difficult to realize *obliviously*, and (3) the high-level ideas ppCDCL uses to overcome these challenges:

$$\begin{aligned}\Phi = & (x_7 \vee x_8) \wedge (\neg x_9 \vee \neg x_8 \vee x_1) \wedge \\ & (x_9 \vee \neg x_2 \vee \neg x_8) \wedge (\neg x_9 \vee \neg x_8 \vee \neg x_1) \wedge \Phi_{extra}\end{aligned}$$

where  $\Phi_{extra}$  involves variables  $x_3, \dots, x_6$ .

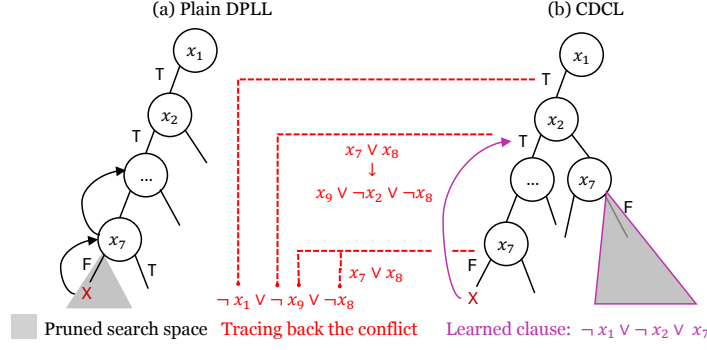


Figure 1: DPLL vs. CDCL with irrelevant decisions ( $x_3, \dots, x_6$  shown as  $\dots$ ). After a conflict, (a) DPLL reacts locally by changing the most recent decision under the current search prefix, whereas (b) CDCL prunes a much larger search space by learning the clause  $\neg x_1 \vee \neg x_2 \vee x_7$ , obtained by tracing the falsified clause back to the relevant decisions that caused the conflict.

## 2.1 Effectiveness of CDCL

Suppose the solver makes the following sequence of decisions:  $x_1=T$ ,  $x_2=T$ ,  $x_3=T$ ,  $x_4=T$ ,  $x_5=T$ ,  $x_6=T$ , and finally  $x_7=F$ .<sup>1</sup> This triggers the following implication chain:

$$x_7=F \text{ forces } x_8=T \text{ due to } (x_7 \vee x_8),$$

and then

$$x_2=T \text{ and } x_8=T \text{ force } x_9=T \text{ due to } (x_9 \vee \neg x_2 \vee \neg x_8).$$

This immediately falsifies  $(\neg x_9 \vee \neg x_8 \vee \neg x_1)$  because  $x_9=T$ ,  $x_8=T$ , and  $x_1=T$ , producing a conflict.

The key observation is that the intermediate decisions  $x_3, \dots, x_6$  are irrelevant: the implication chain and the conflict depend only on  $x_1, x_2$ , and  $x_7$ . Once  $x_1=T$  and  $x_2=T$  are fixed, setting  $x_7=F$  will always reproduce the same propagation and the same conflict, regardless of how  $x_3, \dots, x_6$  are chosen. CDCL is effective precisely because it can identify and exploit this kind of structure (Figure 1).

**Conflict analysis and clause learning.** CDCL learns from conflicts in a systematic way. It first performs *conflict analysis* to identify the assignments responsible for the conflict and records this information as a learned clause. It then *backtracks* to an earlier decision level so that the learned clause can immediately block the same failure pattern. The learned clause is a logical consequence of the original formula, so adding it preserves satisfiability.

In our example, a CDCL solver first constructs the following implication graph:

$$\neg x_7 \xrightarrow{x_7 \vee x_8} x_8 \xrightarrow{x_9 \vee \neg x_2 \vee \neg x_8} x_9 \xrightarrow{\neg x_9 \vee \neg x_8 \vee \neg x_1} \text{conflict}.$$

Next, it derives a learned clause by resolving backward along the edges. Starting with the conflict clause  $(\neg x_9 \vee \neg x_8 \vee \neg x_1)$ , the solver resolves it with the reason clause for  $x_9$ . This process repeats until exactly one current-level literal remains:

$$\begin{aligned} (\neg x_9 \vee \neg x_8 \vee \neg x_1) \otimes (x_9 \vee \neg x_2 \vee \neg x_8) &\Rightarrow (\neg x_8 \vee \neg x_2 \vee \neg x_1), \\ (\neg x_8 \vee \neg x_2 \vee \neg x_1) \otimes (x_7 \vee x_8) &\Rightarrow (\neg x_1 \vee \neg x_2 \vee x_7). \end{aligned}$$

<sup>1</sup>For illustration, we assume that clauses in  $\Phi_{extra}$  do not become unit or cause conflicts at this point.

Here  $\otimes$  denotes *resolution*. The final learned clause,  $(\neg x_1 \vee \neg x_2 \vee x_7)$ , captures the root cause of the conflict: the assignments  $x_1=\text{T}$ ,  $x_2=\text{T}$ , and  $x_7=\text{F}$  are incompatible, regardless of how  $x_3, \dots, x_6$  are assigned. Upon learning this clause, the CDCL solver backtracks to the decision level of  $x_2$ . This single jump undoes the irrelevant assignments to  $x_3, \dots, x_6$  and makes the learned clause unit, immediately forcing  $x_7=\text{T}$ .

**Lazy propagation via two-watched literals.** Modern CDCL solvers also owe much of their speed to *lazy* propagation data structures. A naive propagation loop (as in ppSAT) scans the entire formula after every assignment to check for conflicts or forced literals. In contrast, the two-watched-literal (2WL) scheme allows the solver to ignore the vast majority of clauses on most steps.

In 2WL, the solver maintains two *watched literals* (highlighted in green below) in each clause. After a clause is processed, it either has two non-falsified watched literals, is already satisfied by its retained true watched literal, or has been identified as unit or conflicting. Thus, the solver needs to revisit a clause only when one of its watched literals becomes false. For each literal  $\ell$ , the watchlist  $\text{WL}[\ell]$  stores the clauses currently watching  $\ell$ .

$$(x_7 \vee x_8) \wedge (\neg x_9 \vee \neg x_8 \vee x_1) \wedge (x_9 \vee \neg x_2 \vee \neg x_8) \wedge (\neg x_9 \vee \neg x_8 \vee \neg x_1).$$

In our example, none of  $\neg x_1, \neg x_2, \dots, \neg x_6$  is watched by any listed clause, so the solver does not visit a single listed clause while assigning  $x_1$  through  $x_6$ . This is how 2WL achieves efficiency while maintaining correctness: a clause cannot become unit or conflicting until one of its watched literals is falsified, so all unaffected clauses can be safely ignored. In contrast, scanning the entire formula for propagation, as in ppSAT, performs work proportional to the formula size at every assignment. Continuing with our example, the decisions  $x_1, \dots, x_6$  would cause ppSAT to visit all of the listed clauses even though none of them forces a new assignment or produces a conflict.

Finally, 2WL handles backtracking efficiently. In our example, when the solver backtracks to level 2, no watchlist updates are necessary because backtracking only turns assigned literals back to unassigned, which preserves the watched-literal invariant. In a DPLL-based implementation such as ppSAT, by contrast, the solver repeatedly revisits the formula during propagation and backtracking.

## 2.2 Bringing CDCL to 2PC

The main challenge in bringing CDCL to MPC is to retain the two mechanisms that make modern solvers efficient, clause learning and lazy propagation, while hiding the data-dependent behavior on which both rely. ppCDCL addresses this challenge first at the level of execution control flow. Rather than letting the solver follow its native variable-length control flow, ppCDCL compiles the CDCL loop into a sequence of small, fixed-size steps. Figure 2 depicts the high-level control flow of one ppCDCL step. Each solver step visits exactly one clause and performs a fixed amount of work, making its execution pattern data-independent given the public schedule. A public schedule specifies which subroutines may run at each step, with expensive tasks such as conflict analysis invoked only periodically to amortize cost.

Concretely, step  $t$  consists of two phases: (i) a *clause-selection* phase, which picks a clause  $c^t$  based on the prior state  $\mathcal{S}^{t-1}$ , and (ii) a *clause-handling* phase, which inspects  $c^t$  and conditionally executes the corresponding procedures and updates the solver state. Given this oblivious control flow, the remaining task is to realize each subroutine efficiently. We now give an overview of how ppCDCL realizes clause learning and lazy propagation obliviously.

**Oblivious clause learning in ppCDCL.** Clause learning is difficult to realize obliviously because both its *amount of work* and the size of its output are highly variable and depend on the secret input formula. Naively padding every learned clause to a worst-case bound would be extremely wasteful: a learned clause can in

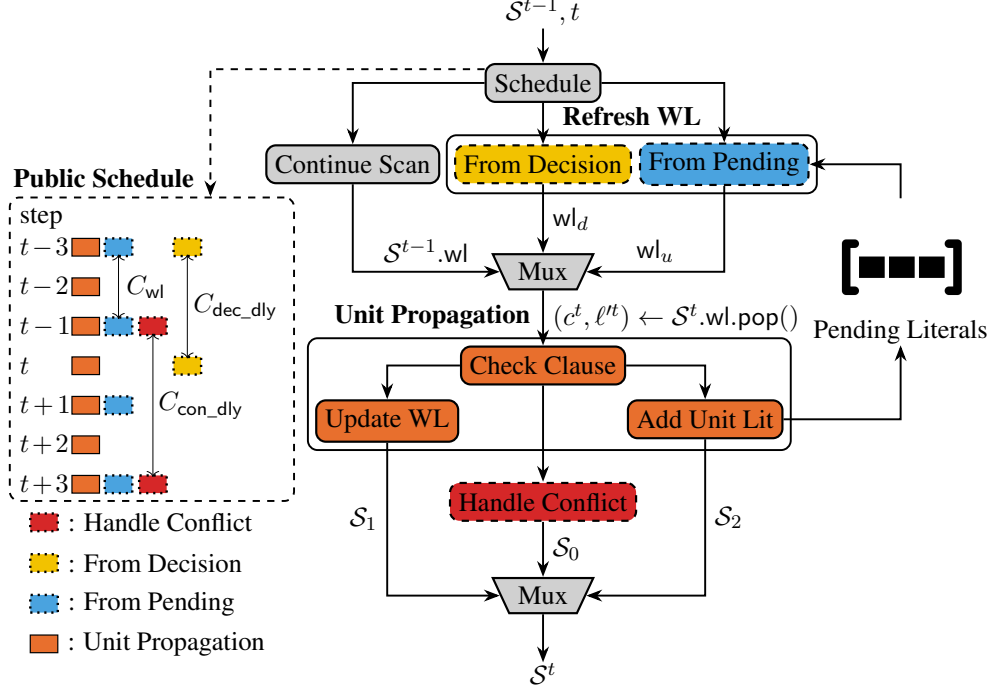


Figure 2: Control flow of one ppCDCL step. At each step, the solver visits one clause  $c^t$  from the current watchlist  $wl$  and conditionally updates the solver state  $\mathcal{S}$ . Branching is handled obviously by a multiplexer: the secret solver state determines whether operations perform real or dummy work. While propagation runs every step, expensive subroutines (dashed rectangles) are enabled only periodically according to a public schedule to amortize cost. If an event is pending off-schedule (e.g., a conflict), the solver performs dummy work until the next scheduled procedure.

principle contain many variables, even though useful learned clauses are typically much shorter. The same problem arises for conflict analysis itself. In our example, the conflict is resolved after only two resolution steps, but worst-case padding would force every conflict to pay for the maximum number of possible steps.

A second difficulty is that conflicts are sparse: most solver steps are propagation or decision steps, not conflict-analysis steps. As a result, the standard oblivious approach of evaluating every branch and multiplexing the result is especially impractical. It would require the solver to execute a full dummy conflict analysis after every step merely to hide whether a real conflict occurred. In our running example, this would mean paying for conflict analysis after every decision  $x_1 = \top, \dots, x_6 = \top$  even though no conflict arises.

ppCDCL makes clause learning feasible by enforcing public budgets on conflict-analysis work and learned-clause size, and by discarding results that exceed either budget. The key observation is that overly long analyses and large learned clauses often have low utility. Discarding such clauses is logically sound, because learned clauses are logically redundant. This design aligns with clause-deletion heuristics in plaintext solvers, which aggressively remove clauses that span many decision levels because such clauses are often less useful for future propagation [AS09]. In our running example, the useful learned clause is powerful precisely because it omits the irrelevant intermediate decisions  $x_3, \dots, x_6$ .

Concretely, in ppCDCL, each conflict analysis runs under a public budget on both the amount of work and the size of the learned clause. If the analysis does not finish within budget, or if the learned clause would exceed the public size bound, ppCDCL discards the partial result. This public-budget mechanism

reduces clause-learning overhead by avoiding worst-case padding in both storage and per-conflict work, while retaining the main pruning benefit of short learned clauses.

**Efficient oblivious propagation in ppCDCL.** Another major challenge in ppCDCL is how to retain the efficiency of lazy data structures such as 2WL in oblivious unit propagation. In 2WL, propagating a literal  $\ell$  requires visiting only the clauses in the watchlist  $WL[-\ell]$ <sup>2</sup>. This selectivity is exactly what makes 2WL efficient in plaintext solving, but also what makes it difficult to realize obliviously. In plaintext execution, most watchlists are short or empty, so many propagation steps cost almost nothing. In our running example, when the solver decides  $x_1 = \top, \dots, x_6 = \top$ , the relevant watchlists are empty, and these propagations incur essentially no cost. In ppCDCL, however, both the watchlist length and the identities of the touched clauses must remain hidden. Padding every watchlist to the total number of clauses would preserve privacy, but it would also destroy the main benefit of 2WL and effectively reduce propagation to a full-formula scan.

ppCDCL preserves the main advantage of 2WL while maintaining obliviousness by combining two ideas: (i) pad each watchlist to a public capacity  $C_{wl}$  that is much smaller than the trivial worst-case bound, and (ii) process exactly *one* watched clause per solver step. In this way, variable-length watchlist scans are compiled into a uniform sequence of micro-steps while preserving the logical behavior of propagation.

The key remaining question is how to choose a safe yet tight value of  $C_{wl}$ . Ignoring clause learning, the watchlist size of a literal is bounded by its occurrence count in the original formula, since each clause contributes to the watchlist of at most one watched literal of a given polarity. In our running example,

$$(x_7 \vee x_8) \wedge (\neg x_9 \vee \neg x_8 \vee x_1) \wedge (x_9 \vee \neg x_2 \vee \neg x_8) \wedge (\neg x_9 \vee \neg x_8 \vee \neg x_1),$$

the most frequent literal is  $\neg x_8$ , which appears three times, so, without learning, a bound of  $C_{wl} = 3$  already suffices. Clause learning can in principle increase watchlist sizes by adding new watched occurrences. The key reason ppCDCL can still use a tight  $C_{wl}$  in practice is that the clause-learning design above filters the cases most likely to create such blow-ups: overly long analyses and overly large learned clauses are discarded before they can accumulate in a single watchlist. This substantially reduces the chance that learned clauses concentrate on a single literal and exceed its watchlist capacity.

Guided by this observation, ppCDCL chooses  $C_{wl}$  using a lightweight *global-occurrence* heuristic: the two parties jointly compute, in a small 2PC circuit, the maximum literal occurrence in their combined formula  $\Phi_A \wedge \Phi_B$ , and use this value as  $C_{wl}$ . This heuristic is efficient, reveals no additional information about the private formula beyond what is implied by  $C_{wl}$ , and in practice produces a tight watchlist bound for the vast majority of instances. As a result, ppCDCL keeps padding overhead low while preserving both privacy and most of the practical benefit of 2WL (Section 6.3).

## 3 Preliminaries

### 3.1 Notation

**SAT and CNF formulas.** A Boolean variable is denoted by  $x_i$  for  $i \in [n]$ , and a *literal* is either  $x_i$  or  $\neg x_i$ . Equivalently, we identify literals with signed variables  $\{\pm x_1, \dots, \pm x_n\}$ , where  $+x_i \equiv x_i$  and  $-x_i \equiv \neg x_i$ . For a literal  $\ell$ , let  $\bar{\ell}$  denote its *complementary literal*; under the signed representation,  $\bar{\ell} = -\ell$ . A (CNF) *clause* is a disjunction  $c = (\ell_1 \vee \dots \vee \ell_k)$ , and a CNF formula  $\Phi$  is a conjunction of clauses, i.e.,  $\Phi = c_1 \wedge \dots \wedge c_m$ . We use  $n$  and  $m$  to denote the number of Boolean variables and clauses in a formula, and use  $C_{\text{phi\_lit}}$  to denote the maximum length of an original clause in  $\Phi$ , i.e.,  $C_{\text{phi\_lit}} = \max_{c \in \Phi} |c|$ . Given a partial assignment,

<sup>2</sup>That is, the watchlist indexed by the literal that becomes falsified when  $\ell$  is set to true.

---

**Algorithm 1:** CDCL SAT Solver

---

```
Input: CNF formula  $\Phi$ 
Output: SAT with assignment  $\alpha$ , or UNSAT
Initialize:  $\alpha \leftarrow \emptyset$  // partial assignment
while true do
   $(\alpha, \text{conflict}) \leftarrow \text{UnitPropagation}(\Phi, \alpha)$ 
  if conflict then
    if no decisions made then
      return UNSAT
     $c \leftarrow \text{ConflictAnalysis}()$  // derive learned clause
     $\Phi \leftarrow \Phi \cup \{c\}$  // add to clause database
     $\alpha \leftarrow \text{Backtrack}(c, \alpha)$  // non-chronological backtracking
  else if all variables assigned then
    return SAT,  $\alpha$ 
  else
     $\ell \leftarrow \text{Decision}()$  // pick unassigned variable
     $\alpha \leftarrow \alpha \cup \{\ell\}$ 
```

---

a literal is *falsified* if it evaluates to **false**, and *non-falsified* if it is either **true** or unassigned. A clause is *conflicting* if all of its literals are falsified, and *unit* if it is not yet satisfied and exactly one literal remains unassigned.

### 3.2 SAT Solving and CDCL

Modern SAT solvers are based on *conflict-driven clause learning* (CDCL), which augments DPLL with (i) learning a clause from each conflict and (ii) *non-chronological backtracking* [MSS99]. Algorithm 1 gives the standard CDCL control flow with three subroutines: unit propagation, conflict analysis, and decision.

**Unit propagation.** Given a partial assignment  $\alpha$ , unit propagation  $\text{UnitPropagation}(\Phi, \alpha)$  repeatedly finds *unit* clauses, i.e., clauses that are not yet satisfied and have exactly one unassigned literal, and forces that literal, recording the clause as its *antecedent* or reason clause. Propagation terminates when no unit clause remains or when some clause becomes fully falsified, producing a *conflict*.

**Conflict analysis, clause learning, and backtracking.** Upon a conflict at decision level  $d$ , CDCL constructs an *implication graph* whose nodes are assigned literals and whose edges point from antecedent literals to the implied literal [MSLM21]. Conflict analysis derives a learned clause by iterated *resolution* over antecedent clauses to obtain a clause that summarizes the assignments responsible for the conflict. The learned clause is added to  $\Phi$ , and the solver *backtracks* to the maximum decision level among the learned clause’s non-asserting literals (or level 0 for a unit learned clause); unit propagation then immediately forces its remaining literal. This non-chronological backtracking is the key departure from DPLL, which backtracks chronologically.

**Decisions.** If propagation halts without conflict and some variables remain unassigned, the solver selects an unassigned variable (and a polarity) and extends  $\alpha$  with a decision literal, continuing the search.

The performance of CDCL depends on how the subroutines are instantiated. State-of-the-art solvers have largely converged on a small set of choices: lazy propagation data structures, 1-UIP learning, VSIDS-style branching with phase saving, and periodic restarts [MMZ<sup>+</sup>01, SE05, PD07, Hua07]. ppCDCL follows the same design choices and adapts them to the privacy-preserving setting.

**Lazy data structures and two-watched literals.** ppCDCL instantiates propagation with the standard *two-watched-literal* (2WL) scheme [MMZ<sup>+</sup>01]. Each clause  $c$  maintains two watched literals, and for every

literal  $\ell$  the solver stores a watchlist  $wl_\ell$  of clauses currently watching  $\ell$ . The key invariant is: if  $c$  is not unit or conflicting, then at least one watched literal of  $c$  is non-falsified; clauses are processed only when a watched literal becomes falsified. Concretely, when  $\ell$  is set to  $\top$ , the solver scans  $wl_\ell$  and, for each clause  $c$  with the other watched literal  $\ell'$ , searches  $c$  for a replacement watch distinct from  $\ell'$ : (i) if such a non-falsified literal  $x$  exists, move the watch from  $\ell$  to  $x$  (update watchlists); (ii) otherwise, if  $\ell'$  is unassigned, then  $c$  is unit and implies  $\ell'$  (recording  $c$  as its reason); (iii) otherwise, if  $\ell'$  is true, then  $c$  is satisfied and no action is needed; (iv) otherwise,  $\ell'$  is falsified and  $c$  is conflicting, triggering conflict analysis.

**1-UIP conflict analysis.** ppCDCL performs ConflictAnalysis using the standard *first unique implication point* (1-UIP) scheme [MSS99]. Let  $d$  be the current decision level and  $\text{level}(\ell)$  the level where  $\ell$  was assigned. Starting from the conflicting clause, 1-UIP repeatedly resolves with antecedent (reason) clauses to eliminate literals at level  $d$  until the learned clause contains exactly one literal  $\ell^*$  with  $\text{level}(\ell^*) = d$ . The learned clause is *asserting*: after backtracking to the maximum decision level among the other literals, unit propagation immediately forces  $\ell^*$ . ppCDCL learns the 1-UIP clause and backtracks to the level induced by it.

**VSIDS decision heuristic with phase saving.** ppCDCL uses VSIDS [MMZ<sup>+</sup>01, LGZ<sup>+</sup>15]. Each variable  $x_i$  has an activity score  $\text{Score}[i]$ ; during conflict analysis, the scores of encountered variables are *bumped*, and periodically all scores are *decayed* so that recent conflicts dominate. When a decision is needed, the solver picks an unassigned variable with maximum activity (ties broken deterministically by index). For polarity, ppCDCL uses *phase saving* [PD07]: it reuses the most recently assigned value of a variable (implied or decided), including across restarts; if a variable has no saved phase, ppCDCL deterministically chooses the negative phase (breaking ties by literal index).

**Restarts.** To avoid spending excessive time in unproductive regions, ppCDCL uses periodic restarts [Hua07, BF15, K LW22] following the *Luby* schedule [LSZ93]. The  $i$ -th restart occurs after  $u \cdot t_i$  (scheduled) conflicts, where  $u$  is a public cost unit and

$$t_i = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1, \\ t_{i-2^{k-1}+1}, & \text{if } 2^{k-1} \leq i < 2^k - 1. \end{cases}$$

For privacy, ppCDCL counts *scheduled conflict events* (real or dummy) as the unit of progress; we set  $u = 256$  so the restart cadence is fully public. On restart, ppCDCL backtracks to decision level 0 (clearing assignments) while retaining learned clauses and heuristic state (activities and saved phases).

### 3.3 Cryptographic Preliminaries

**Distributed point functions.** A distributed point function (DPF) [GI14, BGI16] is a function secret sharing scheme that secretly shares a point function between two parties. A point function  $f_{(\alpha, \beta)}(x)$  evaluates to  $\beta$  only when  $x$  equals  $\alpha$  and to 0 otherwise. With two DPF keys corresponding to  $\alpha$  and  $\beta$ , two parties can locally evaluate the full domain and obtain the secret-shared value of each  $f_{(\alpha, \beta)}(x)$  for all  $x$  in the domain. Recent works [GYW<sup>+</sup>23, ZGY<sup>+</sup>24] allow two parties to jointly evaluate the DPF keys from the secret-shared values  $\alpha$  and  $\beta$ , with  $O(\lambda \cdot \log n + \log |\mathbb{F}|)$  communication and  $O(n)$  computation, where  $n$  is the domain size,  $\lambda$  is the security parameter, and  $\mathbb{F}$  is the range of  $\beta$ .

**RAM-model secure two-party computation.** RAM-model secure two-party computation (RAM-2PC) [GKK<sup>+</sup>12] enables two parties to jointly evaluate a RAM program while ensuring the privacy of their inputs and memory accesses. Unlike standard 2PC, which only evaluates a circuit, RAM-2PC supports dynamic read and write operations on secret-shared memory at private, data-dependent addresses. Floram [Ds17] introduced an efficient Distributed Oblivious RAM (DORAM) construction. Building on Floram, recent

work [ZGY<sup>+</sup>24] extended it to implement a general RAM-2PC protocol in combination with customized garbled circuits.

In ppCDCL, we treat the solver algorithm as a RAM program and execute it with RAM-2PC protocols in the semi-honest model, i.e., we assume both parties follow the protocol and attempt to learn information about each other’s private formula. We discuss our threat model in Section 4.1. We further apply custom optimizations based on the characteristic access patterns of each data structure, which are discussed in Section 4.5.

## 4 The ppCDCL Solver

In this section, we detail the design and construction of ppCDCL. We first state the threat model of ppCDCL in Section 4.1. We then describe the oblivious CDCL control flow built around 2WL propagation (Section 4.2) and our oblivious conflict-analysis and clause-learning design (Section 4.3). Next, we instantiate the oblivious data structures needed for efficient unit propagation (Section 4.4) and describe ORAM-level optimizations tailored to CDCL access patterns (Section 4.5). Finally, we formalize the security guarantee and discuss the explicit leakage of ppCDCL in Section 4.6.

### 4.1 Threat Model

ppCDCL is an interactive two-party protocol between input owners  $P_A$  and  $P_B$ . Each party provides a private CNF formula  $\Phi_i$  after local preprocessing, and the parties jointly evaluate ppCDCL on the conjunction  $\Phi = \Phi_A \wedge \Phi_B$ .

Following ppSAT [LJA<sup>+</sup>22], we consider a static semi-honest adversary that corrupts at most one party. The corrupted party follows the prescribed protocol, but may choose any well-formed input and attempt to infer information about the other party’s formula from its complete protocol view. This view contains its input, randomness, the public inputs and parameters, all messages sent and received, its local protocol state, and the protocol output. In ppCDCL, the subroutine schedule is a deterministic function of the public parameters and the current step, and is therefore public; the number of top-level solver steps executed before termination is also observable. The secret solver state, branch conditions, private memory addresses, and intermediate plaintext values remain hidden subject to the security of the underlying secure-computation and private-memory protocols. We formalize the permitted output and leakage in Section 4.6.

### 4.2 Oblivious Unit Propagation

ppCDCL implements the standard two-watched-literal propagation scheme (Section 3.2) inside an oblivious, step-by-step control flow. To hide both the watchlist length and the clause identities, ppCDCL pads every watchlist to the public capacity  $C_{wl}$  and processes exactly one watched clause per solver step. At step  $t$ , the solver maintains a current propagation literal  $\ell^t$  and a current watchlist  $wl = WL[-\ell^t]$ . Each watchlist entry  $(c^t, \ell^{tt})$  stores a clause ID and that clause’s other, retained watched literal. Depending on the public schedule and the secret solver state, it selects one entry  $(c^t, \ell^{tt})$  from one of three sources:

- *Continue scan*: If the current watchlist scan is not finished (i.e.,  $t \bmod C_{wl} \neq 0$ ), set  $(c^t, \ell^{tt}) \leftarrow wl.pop()$ .
- *Refresh from pending*: Otherwise, pop a pending unit literal  $\ell_u$  from  $U$ , if one exists, refresh the current watchlist by setting  $wl \leftarrow WL[-\ell_u]$ , and set  $(c^t, \ell^{tt}) \leftarrow wl.pop()$  (Algorithm 9).

- *Refresh from decision*: If no pending unit is available, select a decision literal  $\ell_d$ , refresh the current watchlist with  $wl \leftarrow WL[-\ell_d]$ , and set  $(c^t, \ell^{tt}) \leftarrow wl.pop()$  (Algorithm 4).

These sources correspond to the first multiplexer in Figure 2: the solver conditionally refreshes  $wl$  from a pending unit or a decision, and then pops one entry  $(c^t, \ell^{tt})$  from the selected watchlist. The branch condition is secret, so inactive branches perform dummy work. Because every watchlist has the same public capacity  $C_{wl}$ , a full watchlist scan always takes exactly  $C_{wl}$  steps, and refreshes occur only at public scan boundaries  $t \bmod C_{wl} = 0$  (Figure 2)<sup>3</sup>. A natural question is how to select  $C_{wl}$ , as its appropriate value can vary by input instance. As we show later, we design a heuristic that derives a tight  $C_{wl}$  value privately using a small 2PC circuit (Section 5.2).

Given the selected clause  $c^t$  and retained watch  $\ell^{tt}$ , CHECKCLAUSE (Algorithm 5) scans the clause and determines whether propagation should move a watch, enqueue a new implication, report a conflict, or do nothing for a satisfied clause. It outputs candidate watched literals  $(x, y)$  together with secret outcome bits:

- *Watch update* ( $b_{wl} = 1$ ): a replacement non-falsified literal is found, so the watched literals can be updated to  $(x, y)$ .
- *Unit* ( $b_{unit} = 1$ ): no replacement exists and  $\ell^{tt}$  is unassigned, so it must be assigned and enqueued for later propagation.
- *Conflict* ( $b_{conf} = 1$ ): no replacement exists and  $\ell^{tt}$  is falsified, so conflict analysis should be performed at the next scheduled slot.
- *Satisfied/no action* (all three bits are 0 with  $b_{act} = 1$ ): no replacement exists and  $\ell^{tt}$  is true. With  $b_{act} = 0$ , the same all-zero bits denote dummy/no action.

These bits determine whether the following subroutines perform real work or dummy work:

- *Update WL* (Algorithm 10): move  $c^t$  to the new watched literal, or perform dummy work if no update is needed.
- *Add Unit Lit.* (Algorithm 8): assign and enqueue the implied literal, or perform dummy work if the clause is not unit.
- *Handle Conflict* (Algorithm 12): perform conflict analysis and backtracking, as described next.

### 4.3 Oblivious Clause Learning

**Combining clause learning with clause deletion.** As discussed in Section 2.2, oblivious clause learning is difficult because both data-structure sizes and per-conflict work vary widely; padding to worst-case bounds would negate the benefit of learning.

Our key insight is to combine clause learning with a deletion policy that is simple enough to make oblivious. Clause length and the number of resolution steps serve as publicly bounded proxies for learned-clause utility: very long learned clauses or analyses that require many resolution steps are expensive in MPC and often provide limited propagation power. This design is inspired by clause-deletion heuristics in plaintext solvers [MSS99, SE05, AS09, AGCLS15], which aggressively remove clauses judged unlikely to help future propagation. Accordingly, ppCDCL treats over-budget learning outcomes as clauses to be

---

<sup>3</sup>Note that the more expensive decision procedure follows a different schedule; we discuss this design choice at the end of Section 4.3.

deleted immediately. Concretely, conflict analysis in ppCDCL is parameterized by three public bounds: (i) a maximum learned-clause length  $C_{\text{con\_lit}}$ , (ii) a public append capacity  $C_{\text{con\_num}}$  for valid or dummy learned-clause entries, and (iii) a cap  $C_{\text{uiip}}$  on resolution steps. Every stored learned clause is padded to length  $C_{\text{con\_lit}}$ , and the learned-clause database stores at most  $C_{\text{con\_num}}$  entries. If the learned clause would exceed  $C_{\text{con\_lit}}$ , the public append region is exhausted, or the analysis would exceed  $C_{\text{uiip}}$  resolution steps, then ppCDCL discards the partial result and restarts from decision level 0<sup>4</sup>.

This design reduces clause-learning overhead while integrating clause deletion at no additional asymptotic cost. The deletion itself does not introduce additional control-flow leakage: conflict analysis executes under the same public budget, and the subsequent restart is indistinguishable from a regular backtrack; any effect on termination is captured by  $T$ , which we discuss in Section 4.6. In Section 6.3, we show that these budgets can be set far below worst-case bounds and can significantly improve end-to-end runtime.

**A public schedule with delay.** Although the bounded design above makes clause learning feasible, conflict analysis remains one of the most expensive ppCDCL components because it performs substantial linear work: resolution scans clauses, updates heuristic state, and backtracking scans variable state to undo assignments. At the same time, decisions and conflicts occur much less frequently than propagation steps. A naive oblivious implementation that enables decision and conflict-analysis blocks at every step would therefore waste most of its work on dummy executions.

ppCDCL instead decouples conflict detection from conflict analysis and executes conflict analysis and decision subroutines only periodically according to a *public delayed schedule*. Concretely, ppCDCL enables decision and conflict analysis after every  $C_{\text{dec\_dly}}$  and  $C_{\text{con\_dly}}$  propagation steps, respectively, where  $C_{\text{dec\_dly}}$  and  $C_{\text{con\_dly}}$  are configurable public parameters. At step  $t$ , the solver computes  $\{b_{\text{dec}}, b_{\text{con}}, b_u\} \leftarrow \text{Schedule}(t, C_{\text{wl}}, C_{\text{dec\_dly}}, C_{\text{con\_dly}})$ , where  $b_u = 1$  iff  $t \bmod C_{\text{wl}} = 0$ ,  $b_{\text{dec}} = 1$  iff  $t \bmod C_{\text{dec\_dly}} = 0$ , and  $b_{\text{con}} = 1$  iff  $t \bmod C_{\text{con\_dly}} = 0$ . These public bits determine when the solver may refresh the current watchlist, make a decision, or execute conflict analysis. If an operation is pending but not enabled, ppCDCL preserves the pending state and performs dummy work until the next scheduled opportunity. This hides the secret solver state within each scheduled step while amortizing expensive operations over many propagation steps. As we show in Section 6.3, the delayed schedule yields substantial runtime improvements over enabling decision and conflict-analysis procedures at every step.

## 4.4 Oblivious Lazy Data Structures

Section 4.2 describes the oblivious control flow of ppCDCL while treating the solver state abstractly. We now instantiate these data structures concretely. We begin by defining the basic types (literals, clauses, etc.), and then explain how each type is stored and accessed via ORAM-based oblivious lazy data structures.

**Clauses, formulas, and watchlists.** We represent literals, clause indices, and clause contents using fixed-width bit strings. Their widths are determined by the public variable bound  $n$ , the original-clause length and count bounds  $C_{\text{phi\_lit}}$  and  $m$ , the learned-clause length and count bounds  $C_{\text{con\_lit}}$  and  $C_{\text{con\_num}}$ , and the watchlist-size bound  $C_{\text{wl}}$ .

- *Literals.* A literal is a signed integer of width  $w_{\text{lit}} = \lceil \log_2(n+1) \rceil + 1$ , with  $i > 0$  denoting  $\ell$ ,  $-i$  denoting  $-\ell$ , and 0 reserved as a dummy padding symbol.
- *Clauses.* A clause is encoded as a fixed-length vector with a total of  $\max(C_{\text{con\_lit}}, C_{\text{phi\_lit}})$  literals, padded with dummy literal 0.

---

<sup>4</sup>In principle, the solver could also backtrack chronologically. We adopt restart because it is consistently faster in our experiments; we discuss the trade-off in Appendix B.1.

- *Clause index.* A clause index  $c_{idx}$  is a signed integer whose sign distinguishes original vs. learned clauses and whose magnitude is the logical clause id. Concretely,  $c_{idx} \in [-C_{con\_num}, m]$  with 0 reserved as a dummy index. We represent its width with  $w_{cid} = 1 + \lceil \log_2(\max(C_{con\_num}, m) + 1) \rceil$ .
- *Watchlists.* A watchlist  $wl$  is a fixed-length vector of  $C_{wl}$  entries  $(c_{idx}, \ell_{other})$ , padded with  $(0, 0)$ ; its width is  $w_{wl} = C_{wl} \cdot (w_{cid} + w_{lit})$ . If clause  $c$  watches  $(a, b)$ , then  $WL[a]$  contains  $(c, b)$  and  $WL[b]$  contains  $(c, a)$ .

**Oblivious lazy data structures.** We now describe how ppCDCL stores and accesses these basic types obliviously. We partition the data structure into two categories: (i) large, frequently accessed structures are stored in ORAM (clauses, watchlists, and per-variable state), and (ii) lightweight structures that are accessed only in infrequent subroutines are stored as plain arrays, since a linear scan for oblivious access is cheaper than maintaining a dedicated ORAM. Concretely, the data structures in ppCDCL are:

- *Original-clause ORAM*  $CL_\phi$ : stores the  $m$  original clauses in  $\Phi$ . Each clause is stored as a bit string of length  $w_{lit} \cdot C_{phi\_lit}$ .
- *Learned-clause ORAM*  $CL_{con}$ : stores up to  $C_{con\_num}$  learned conflict clauses. Each learned clause is stored as a bit string of length  $w_{lit} \cdot C_{con\_lit}$ .
- *Watchlist ORAM*  $WL$ : stores the watchlists for all literals. There are  $2n$  watchlists, one per polarity, and each watchlist is a bit string of length  $C_{wl} \cdot (w_{cid} + w_{lit})$ .
- *Variable-state ORAMs*  $V$ : store the following state fields for all  $n$  variables: (i) assignment, (ii) assignment order, (iii) decision level, (iv) antecedent clause index, and (v) phase-saving polarity. Assignments use 2 bits (unassigned/false/true), decision levels use  $\lceil \log_2(n+1) \rceil$  bits, antecedents use  $w_{cid}$  bits, and phase-saving polarities use 1 bit.
- *Pending Literal Array*  $U$ : a length- $n$  array where  $U[i] \in \{0, 1, 2\}$  encodes whether  $-i$  is pending (1),  $i$  is pending (2), or neither (0).
- *Activity Score Array*  $Score$ : a length- $n$  array storing fixed-point activity scores. During conflict analysis, the scores of encountered variables are increased by 100, and after each conflict all scores are decayed according to

$$Score[i] \leftarrow \lfloor 95Score[i]/100 \rfloor \quad \text{for every } i \in [n].$$

It suffices to represent each score using  $\lceil \log_2(2001) \rceil = 11$  bits, since a variable bumped in every conflict has score at most  $100/(1 - 0.95) = 2000$ ; we use 16 bits in our implementation.

## 4.5 ORAM Optimizations

The ORAMs used for clauses, watchlists, and variable states support generic reads and writes, which suffice for the required operations. However, each of these data structures exhibits distinct access patterns that we can leverage to design special-purpose ORAM implementations and reduce overhead. We first summarize these access patterns.

**Access patterns.** *Clauses.* We separate clause storage to exploit the different access patterns. Since the solver never modifies original clauses, we place them in a dedicated *read-only* ORAM  $CL_\phi$ . Similarly, the solver never modifies existing entries in the learned-clause ORAM  $CL_{con}$  but might perform append-only writes: the solver deterministically writes to  $CL_{con}$  every  $C_{con\_dly}$  steps, and thus the write index can be made public. We therefore maintain a *public learned-slot index*: at each scheduled conflict event (see Section 4.2), the solver performs either a valid write or a dummy write at the current slot. The index advances through slots  $1, \dots, C_{con\_num}$  and then remains saturated at  $C_{con\_num} + 1$ , where accesses use dummy ID 0.

Because in-range dummy writes also consume slots,  $C_{\text{con\_num}}$  is the append capacity of  $\text{CL}_{\text{con}}$  and therefore upper-bounds the number of retained learned clauses.

*Watchlists.* The watchlist ORAM WL stores watchlists for all literals, where each logical element is a pair containing a clause index and the corresponding clause’s other watched literal. The solver either retrieves a full watchlist via a *range read* of  $C_{\text{wl}}$  consecutive entries (starting from a secret index), or performs a constant number of individual-entry updates; UPDATEWL performs three such fixed-scan mutations.

*Variable state.* The variable-state ORAMs V maintain solving progress (assignment, decision level, antecedent, and phase saving). These are accessed via standard private reads and writes throughout propagation, decision, and conflict analysis; we do not apply structure-specific optimizations to these general-purpose accesses.

The concrete ORAM operations used by ppCDCL are:

- *Original-clause ORAM*  $\text{CL}_\phi$ : read-only access in CHECKCLAUSE.
- *Learned-clause ORAM*  $\text{CL}_{\text{con}}$ : append-only writes in CONFLICTANALYSIS; private reads in CHECKCLAUSE and CONFLICTANALYSIS.
- *Watchlist ORAM* WL: public-length range reads to load an entire watchlist  $\text{WL}[\ell]$  in DECISION and GETPENDINGUNITLITERAL; individual private writes in UPDATEWL.
- *Variable-state ORAMs* V: standard private reads/writes.

Next, we detail our optimizations that exploit the read-only, append-only, and range-access structure above to further reduce ORAM overhead.

**Floram overview.** To implement an ORAM in secure computation, Floram [Ds17] uses a read-only memory (ROM), a write-only memory (WOM), and a stash (S) that supports both read and write operations. Specifically, ROM is the encrypted version of the memory and is identical for both parties. WOM is a secret-shared version of the memory and is additively shared between the two parties. The protocol guarantees the following: (1) the WOM always contains the most recent data but cannot be read directly; (2) the ROM and S together contain the most recent data, with the version in S taking priority. Initially, data is stored in both ROM and WOM, and S is empty. For a read operation at index  $i$ , the two parties jointly generate DPF keys corresponding to  $\alpha = i$  and  $\beta = 1$  and locally evaluate the DPF over the full domain to obtain a secret-shared vector with value 1 at index  $i$  and 0 elsewhere. Then they compute the inner product of the vector with the ROM to obtain the secret sharing of the data at index  $i$  in the ROM. They further linearly scan S to potentially obtain the most recent version. For a write operation at index  $i$ , the two parties first perform the same read operation to obtain the secret sharing of the data at index  $i$  and compute the difference  $\delta$  between the new value and the current value. Then they generate DPF keys corresponding to  $\alpha = i$  and  $\beta = \delta$  and locally evaluate the DPF over the full domain to derive a secret-shared vector with value  $\delta$  at index  $i$  and 0 elsewhere. They add this vector to their secret-shared WOM to update the data at index  $i$  and append the new version to S. When S is full, the two parties run a refresh protocol to update ROM from WOM and empty S.

We follow the blueprint above and propose the following optimizations for each specific data structure:

- *Original-clause ORAM*  $\text{CL}_\phi$ : This ORAM supports only read operations. In our construction, we drop S and WOM. We perform read operations by fetching data from ROM using DPF. This optimization eliminates the overhead of linearly scanning S and refreshing ROM.
- *Learned-clause ORAM*  $\text{CL}_{\text{con}}$ : This ORAM supports read and append-only write operations. In our construction, we use the same approach as the read-only ORAM to perform read operations. For each write operation, we directly update the corresponding entry in ROM since the index is publicly known.
- *Watchlist ORAM* WL: This ORAM supports typical ORAM operations as well as public-length range reads. We use the same construction for read and write operations as Floram. We note that the DPF full

### Functionality $\mathcal{F}_{\text{ppcdcl}}$

**Public input.** Both parties provide a common public solver configuration

$$p = (n, m, C_{\text{phi\_lit}}, \theta, \text{max\_t}),$$

where  $(n, m, C_{\text{phi\_lit}})$  are the CNF size parameters,  $\theta = (C_{\text{wl}}, C_{\text{con\_lit}}, C_{\text{con\_num}}, C_{\text{uip}}, C_{\text{con\_dly}}, C_{\text{dec\_dly}})$  is the tuple of solver parameters, and  $\text{max\_t}$  is the step budget.

**Private input.** Each party  $P_i$ , for  $i \in \{A, B\}$ , provides an encoded CNF formula  $\Phi_i$ , after any local preprocessing, that is well-formed under the agreed CNF size parameters. Let  $\Phi$  denote the encoding of  $\Phi_A \wedge \Phi_B$ .

**SAT solving.** Execute plaintext ppCDCL on  $\Phi$  under the public configuration  $p$ . Let  $T \in \{0, \dots, \text{max\_t}\}$  be the number of top-level solver steps executed;  $T = 0$  is permitted when the execution terminates before entering the main solver loop. After each executed step, stop if ppCDCL obtains a SAT or UNSAT result or detects a watchlist-capacity overflow. If no such terminal event occurs after  $\text{max\_t}$  steps, return TIMEOUT. Set

$$\text{out} \in \{(\text{SAT}, \alpha), \text{UNSAT}, \text{TIMEOUT}, \text{FAILURE}\},$$

where  $\alpha \models \Phi$  in the SAT case and FAILURE denotes a detected watchlist-capacity overflow.

**Output.** Send out and  $T$  to both parties.

Figure 3: Ideal functionality of ppCDCL.

domain evaluation vector corresponding to  $\alpha + 1$  and  $\beta$  is indeed that corresponding to  $\alpha$  and  $\beta$  shifted by 1. Therefore, for a range-read operation of length  $\ell$  starting at index  $i$ , the two parties generate only one pair of DPF keys, with  $\alpha = i$  and  $\beta = 1$ , and evaluate the secret-shared vector over the full domain. They then shift the vector by 1 to derive vectors for each of the remaining  $\ell - 1$  indices. This optimization reduces the cost of generating DPF keys and evaluating the full domain by a factor of  $\ell$ .

## 4.6 Security and Leakage Profile

Leakage in ppCDCL consists of the solver output, the public formula dimensions, solver parameters, and the realized number of solver steps. We now formalize these quantities and our security claims and discuss each leakage channel in detail.

**Ideal functionality.** Figure 3 shows the ideal functionality of ppCDCL  $\mathcal{F}_{\text{ppcdcl}}$ . For a fixed public configuration, we use *plaintext ppCDCL* to mean the execution of the ppCDCL algorithm over plaintext values and memory, with all operations replaced by their corresponding plaintext operations.

In particular, the functionality takes as public input the solver parameters  $\theta = (C_{\text{wl}}, C_{\text{con\_lit}}, C_{\text{con\_num}}, C_{\text{uip}}, C_{\text{con\_dly}}, C_{\text{dec\_dly}})$ , the joint CNF size parameters  $n, m, C_{\text{phi\_lit}}$ , and the step budget  $\text{max\_t}$ . Given the parties' private formulas  $\Phi_A$  and  $\Phi_B$  after local preprocessing, the functionality runs plaintext ppCDCL on  $\Phi = \Phi_A \wedge \Phi_B$  with the public input parameters  $\theta$  for at most  $\text{max\_t}$  steps and returns the solver output out and step count  $T$  to both parties. We define ppCDCL's leakage function for the public configuration  $p = (n, m, C_{\text{phi\_lit}}, \theta, \text{max\_t})$  as

$$\mathcal{L}(\Phi_A, \Phi_B; p) = (p, \text{out}, T).$$

**Security claims.** Because every operation of ppCDCL is executed inside generic semi-honest components invoked on a public schedule, its security reduces to the security of these components via sequential composition [Can00]. Assuming that the 2PC and ORAM protocols instantiating ppCDCL securely realize

their respective ideal functionalities against a static semi-honest adversary, the resulting ppCDCL protocol  $\Pi$  securely realizes  $\mathcal{F}_{\text{ppcdcl}}$ . Specifically, for each  $i \in \{A, B\}$ , there exists a probabilistic polynomial-time simulator  $\text{Sim}_i$  such that, for every public configuration  $p$  and all well-formed inputs  $(\Phi_A, \Phi_B)$ ,

$$\text{View}_{P_i}^{\Pi}(\Phi_A, \Phi_B; p) \approx_c \text{Sim}_i(\Phi_i, \mathcal{L}(\Phi_A, \Phi_B; p)).$$

Intuitively, the public configuration  $p$  and the step count  $T$  deterministically fix the protocol’s entire execution schedule. The private formulas only affect the secret branch conditions, intermediate solver states, ORAM access indices, and the secret bits determining whether a scheduled subroutine performs real or dummy work. Because all such data-dependent operations are evaluated entirely within the underlying secure components, the complete protocol transcript can be simulated by replaying the public schedule for  $T$  steps and invoking the respective component simulators. Consequently, the execution reveals no information to either party beyond the defined leakage  $\mathcal{L}(\Phi_A, \Phi_B; p)$ .

**Output leakage.** The solver output can itself reveal information about the other party’s formula. A SAT or UNSAT output reveals whether the submitted conjunction is satisfiable and, in the SAT case, the satisfying assignment may reveal additional information about either party’s formula. Moreover, because a party may choose any well-formed input, it can use its formula to ask restricted satisfiability questions about the other formula. For example, choosing the unit-clause formula  $\Phi_A = (x_i)$  can reveal whether  $\Phi_B$  is satisfiable subject to  $x_i = \top$  when the execution returns SAT or UNSAT. Across multiple executions, a party may vary its input and combine the resulting answers; each execution reveals its corresponding leakage  $\mathcal{L}(\Phi_A, \Phi_B; p)$ . Such inference follows from the selected inputs and prescribed outputs and is captured by  $\mathcal{L}(\Phi_A, \Phi_B; p)$ .

**Leakage in formula size.** The values  $n$ ,  $m$ , and  $C_{\text{phi\_lit}}$  are public bounds for the joint CNF. When they are chosen as the exact formula dimensions, they reveal the size and shape of the joint instance; because each party knows its own contribution, they may also constrain the possible size of the other party’s formula. The parties can instead pad their formulas to larger public bounds, at the cost of increased memory use and per-step work.

**Leakage via public solver parameters.** Among all the solver parameters,  $C_{\text{wl}}$  is derived from the private formulas by the global-occurrence heuristic described in Section 5.2 and therefore reveals instance-dependent structural information. In the initial execution, it equals the maximum combined occurrence count of any literal in the joint formula. Together with its own occurrence counts, a party can derive upper bounds on the other party’s occurrence count for each literal. This leakage can be reduced by padding  $C_{\text{wl}}$  upward or selecting it from a conservative public bound, at the cost of longer watchlist scans and higher memory use.

The remaining solver parameters are public configuration choices. In our evaluation, they are fixed per benchmark suite and are not derived from the secret clause contents. The phase schedule is a deterministic function of these parameters and the current step. Consequently, observing scheduled decision or conflict-analysis phases reveals no information beyond these public values; whether a scheduled phase performs real or dummy work remains private.

**Leakage at termination.** The value  $T$  is the number of steps of the top-level ppCDCL loop in Algorithm 3. Because ppCDCL will return the solver output as soon as the formula is solved, different formulas may produce different values of  $T$ , so this leakage can reveal information correlated with the solver’s search behavior or the difficulty of the joint instance.

A fixed-budget variant of ppCDCL can hide the exact value of  $T$  by continuing with dummy steps until  $\text{max\_t}$  after reaching a terminal state. This still reveals the coarser fact of whether a SAT or UNSAT result was obtained within the budget, through the distinction between those outputs and TIMEOUT. Hiding even this distinction for all inputs would require a sufficiently large worst-case budget, which is generally impractical for SAT solving.

| Parameter                                         | Impact |     |     |
|---------------------------------------------------|--------|-----|-----|
|                                                   | FR     | NoS | CPS |
| $C_{\text{uip}}$ (Resolution-Step Cap)            | No     | Yes | Yes |
| $C_{\text{con\_lit}}$ (Learned-Clause Length Cap) | No     | Yes | Yes |
| $C_{\text{con\_num}}$ (Learned-Clause Capacity)   | No     | Yes | Yes |
| $C_{\text{wl}}$ (Watchlist Capacity)              | Yes    | Yes | Yes |
| $C_{\text{dec\_dly}}$ (Decision Schedule)         | No     | Yes | Yes |
| $C_{\text{con\_dly}}$ (Conflict Schedule)         | No     | Yes | Yes |

Table 1: Impact of the parameters on solver performance. Each cell indicates whether the parameter affects the failure rate (**FR**), number of steps (**NoS**), or cost per step (**CPS**).

**Leakage–efficiency trade-offs.** Padding the formula size or  $C_{\text{wl}}$  and executing to a fixed public step budget can reduce the corresponding leakage at the cost of greater per-step work and end-to-end runtime. Following ppSAT, our implementation prioritizes efficiency: it uses the global-occurrence value for  $C_{\text{wl}}$ , makes the resulting leakage explicit, and terminates upon reaching an outcome. This flexibility makes ppCDCL tunable to the desired privacy–efficiency point.

## 5 Parameter Selection and Heuristics

Recall that ppCDCL executes CDCL under a public schedule and uses a small set of public parameters to bound storage, per-step work, and the frequency of expensive subroutines. These parameters affect three aspects of performance: the cost per step (CPS), the total number of steps required to solve a formula (NoS), and the failure rate (FR). The first two determine overall runtime when solving succeeds, while FR captures configuration failures such as overflowing a padded watchlist and rerunning with a larger configuration. Table 1 summarizes the effect of each parameter.

In the remainder of this section, we present a principled parameter-selection strategy. We first introduce the parameters that govern the conflict-analysis scheme in ppCDCL. These parameters bound the amount of work and the sizes of data structures but do not affect solver soundness: because learned clauses are logically redundant, ppCDCL can safely discard them. We then turn to the watchlist-capacity parameter  $C_{\text{wl}}$ , which is the only parameter that can cause overflow and thus directly impacts FR. Importantly, our conflict-analysis design limits how much clause learning perturbs the watchlist distribution, which in turn enables a tight choice of  $C_{\text{wl}}$ . Finally, we discuss the delayed schedule parameters  $C_{\text{dec\_dly}}$  and  $C_{\text{con\_dly}}$ .

### 5.1 Conflict Analysis Parameters

The conflict-analysis scheme in ppCDCL is parameterized by three public bounds:  $C_{\text{uip}}$ ,  $C_{\text{con\_lit}}$ , and  $C_{\text{con\_num}}$ . The parameter  $C_{\text{uip}}$  caps the maximum number of resolution steps performed per conflict,  $C_{\text{con\_lit}}$  caps the length of each retained learned clause (and thus the padded length of all learned clauses), and  $C_{\text{con\_num}}$  caps the public scheduled append region, including valid and dummy entries, thereby upper-bounding the number of retained learned clauses. Together, they determine the amount of work performed during conflict analysis and the ORAM size used to store learned clauses, directly affecting the cost of learning as well as the cost of reading learned clauses during propagation, and therefore CPS. Critically, these

bounds do not affect solver correctness: learned clauses are logically redundant, and ppCDCL can safely discard any lengthy clause or unfinished conflict analysis.

**Choosing the learned-clause length cap  $C_{\text{con\_lit}}$ .** ppCDCL uses a small  $C_{\text{con\_lit}}$  to limit the cost of accessing learned clauses during propagation. Learned clauses exceeding  $C_{\text{con\_lit}}$  are discarded, and the solver restarts from decision level 0. However, a value too small could discard useful learned clauses, weakening the pruning effect of clause learning and increasing NoS.

Our experiments show  $C_{\text{con\_lit}}$  can be set instance-independently and aggressively small. Long learned clauses are expensive to store and scan, yet provide weak pruning power. Modern SAT solvers routinely discard low-quality learned clauses using similar heuristics. In ppCDCL, clause length serves as a simple but effective proxy for utility: a small  $C_{\text{con\_lit}}$  retains only short, high-utility clauses while reducing cryptographic overhead. We show that a small  $C_{\text{con\_lit}}$  results in consistently improved runtime in Section 6.3.

**Choosing the resolution-step cap  $C_{\text{uiip}}$ .** The cap  $C_{\text{uiip}}$  bounds the work of conflict analysis by limiting the number of resolution steps performed per conflict. In ppCDCL, if conflict analysis exceeds  $C_{\text{uiip}}$ , the solver discards the partial result and restarts, avoiding worst-case padding while preserving privacy. Empirically, we find that  $C_{\text{uiip}}$  can be set to a moderate, instance-independent value: as with a small  $C_{\text{con\_lit}}$ , a small  $C_{\text{uiip}}$  reduces expensive resolution work and often *improves* end-to-end runtime by preventing the solver from spending effort on long, low-yield analyses.

**Choosing the learned-clause capacity  $C_{\text{con\_num}}$ .** Precisely predicting the required learned-clause capacity is difficult, as it varies by instance. The public append index advances at every scheduled conflict event until the append region is exhausted, then remains saturated at  $C_{\text{con\_num}} + 1$ ; thus, the capacity  $C_{\text{con\_num}}$  counts both valid and dummy entries and only upper-bounds the number of retained learned clauses. Like all conflict-analysis parameters,  $C_{\text{con\_num}}$  does not affect correctness: learned clauses are logically redundant and new clauses are safely discarded after the append region is exhausted. Additionally,  $C_{\text{con\_num}}$  affects CPS only through learned-clause ORAM size, and empirically, a large  $C_{\text{con\_num}}$  incurs modest overhead. We therefore fix  $C_{\text{con\_num}}$  to a reasonably generous value, preserving pruning effectiveness, avoiding a significant increase in NoS, and keeping CPS low.

## 5.2 Watchlist Capacity $C_{\text{wl}}$

Watchlist capacity  $C_{\text{wl}}$  bounds the maximum number of clauses stored in every literal’s watchlist. Since watchlists are padded to length  $C_{\text{wl}}$ , it directly determines the number of steps required to scan a watchlist and thus the overhead of unit propagation, affecting both CPS and NoS (Table 1). A worst-case bound using the total number of clauses is always sufficient, but can be orders of magnitude larger than necessary. Conversely, underestimating  $C_{\text{wl}}$  can cause overflow and halt the solver, increasing FR.

Let  $\text{Lit}(\Phi)$  denote the set of literals in  $\Phi$ , let  $\text{occ}(\Phi, \ell)$  be the number of clauses containing  $\ell$ , and let  $C_{\text{wl}}^*$  be the actual maximum watchlist size realized during execution. Without learning, a safe bound is the maximum occurrence:  $C_{\text{wl}}^* \leq \max_{\ell \in \text{Lit}(\Phi)} \text{occ}(\Phi, \ell)$ . However, with clause learning, a single literal could appear in every retained learned clause. Thus,

$$C_{\text{wl}}^* \leq \max_{\ell} \text{occ}(\Phi, \ell) + C_{\text{con\_num}},$$

so setting  $C_{\text{wl}} = \max_{\ell} \text{occ}(\Phi, \ell) + C_{\text{con\_num}}$  is always safe. In practice,  $C_{\text{con\_num}}$  must be large to retain useful learned clauses, which loosens the bound on  $C_{\text{wl}}$  and incurs unnecessary propagation overhead.

In practice, however, our conflict-analysis scheme substantially reduces the likelihood of such a worst case: learned clauses are short (capped by  $C_{\text{con\_lit}}$ ), per-conflict analysis is bounded (by  $C_{\text{uiip}}$ ), and any

lengthy clauses are discarded. As a result, learned clauses are less likely to dominate the watchlist distribution, and the maximum watchlist size is typically well-approximated by the occurrence profile of the input formula. This observation motivates estimating  $C_{wl}$  using the *global occurrence* of the joint formula  $\Phi_A \wedge \Phi_B$ .

**Global occurrence heuristic.** The two parties compute the maximum occurrence in the joint formula  $\Phi_A \wedge \Phi_B$  directly in 2PC and set

$$C_{wl} = \max_{\ell \in \text{Lit}(\Phi_A \wedge \Phi_B)} \text{occ}(\Phi_A \wedge \Phi_B, \ell).$$

Specifically, both parties first agree on the total number of variables in their joint formula and encode the local occurrence counts of all literals into vectors  $\mathbf{o}_A$  and  $\mathbf{o}_B$ , i.e.,  $\mathbf{o}_i[\ell] = \text{occ}(\Phi_i, \ell)$ . The parties then evaluate a small 2PC circuit to obtain  $\|\mathbf{o}_A + \mathbf{o}_B\|_\infty$  as a public output and set  $C_{wl}$  accordingly. Although global occurrence is a heuristic, we find that it provides a remarkably tight estimate that remains safe (i.e., does not overflow) for the vast majority of instances. This tightness minimizes padding overhead, and the cost ( $< 0.5$  s) is negligible compared with the solving time. We evaluate the failure rate and precision of this heuristic in Section 6.3.

While rare under our global occurrence heuristic, overflow can occur if learned clauses heavily cluster around specific literals, especially when the initial  $C_{wl}$  and the total number of variables are small. If a failure occurs, the parties must restart the protocol with a larger  $C'_{wl}$ . In principle, one could design a more sophisticated restart policy that adapts  $C'_{wl}$  based on observed learning dynamics, but in practice we use a simple, conservative rule: upon FAILURE, rerun with  $C'_{wl} = 10C_{wl}$  and report the sum of both runs. This policy is sufficient in our evaluation: ppCDCL encounters no failure except `pret`, which succeeds on the second attempt.

### 5.3 Delay Schedule $C_{con\_dly}$ and $C_{dec\_dly}$

ppCDCL amortizes the cost of expensive decision and conflict analysis operations by executing them periodically rather than at every step: the solver performs the decision block only at steps where  $step \equiv 0 \pmod{C_{dec\_dly}}$ , and executes conflict analysis and backtracking only when  $step \equiv 0 \pmod{C_{con\_dly}}$ . Together,  $C_{dec\_dly}$  and  $C_{con\_dly}$  affect CPS and NoS but have no impact on FR. In principle, these parameters could be tuned using instance-dependent heuristics or dynamically scheduled (e.g., gradually increasing  $C_{con\_dly}$  over time). In Section 6.3, we demonstrate that fixed, moderate values suffice to achieve significant and consistent runtime improvements.

## 6 Implementation and Evaluation

**Testbed.** We implement ppCDCL using the semi-honest 2PC library from `EMP-toolkit` [WMK16] and adopt our ORAM scheme based on Floram [Ds17]. All experiments run on machines equipped with Apple M1 Pro CPUs and 16 GB of RAM, using 4 threads and a network bandwidth of up to 10 Gbps.

**CNF instance selection.** We evaluate ppCDCL on SATLIB [HS00] to cover diverse SAT domains, and include the smaller Haplotype instances [MCP<sup>+</sup>06, LMS06a, LMS06b] used in ppSAT [LJA<sup>+</sup>22] for direct comparisons. Instance details are in Table 2.

**Experiment setup.** We obtain the two-party inputs via a balanced 50/50 random split on each benchmark instance, repeating the partitioning 50 times and reporting averages. Each party locally preprocesses its input using CaDiBack [BFW23] (Algorithm 2); for fairness, we run ppSAT on both preprocessed and original instances and report the faster result. We set  $C_{wl}$  using the global occurrence heuristic (Section 5.2) and

| Domain                              | Name (#instances)   | #Var $\times$ #Clause |
|-------------------------------------|---------------------|-----------------------|
| <i>Haplotype</i>                    | n4 $ G  = 4$ (29)   | $348 \times 2600$     |
|                                     | n5 $ G  = 5$ (29)   | $475 \times 4062$     |
|                                     | n6 $ G  = 6$ (29)   | $618 \times 5993$     |
|                                     | n7 $ G  = 7$ (29)   | $777 \times 7962$     |
|                                     | n8 $ G  = 8$ (29)   | $951 \times 10399$    |
|                                     | n10 $ G  = 10$ (29) | $1349 \times 16648$   |
| <i>Planning</i>                     | blocks (7)          | $1644 \times 29878$   |
|                                     | logistics (4)       | $1881 \times 11682$   |
| <i>Circuit Analysis</i>             | bf (4)              | $1698 \times 5162$    |
|                                     | ssa (8)             | $2351 \times 6708$    |
| <i>Graph Coloring</i>               | flat100 (100)       | $300 \times 1117$     |
|                                     | flat200 (100)       | $600 \times 2237$     |
|                                     | pret (8)            | $105 \times 280$      |
| <i>Combinatorial &amp; learning</i> | QG (22)             | $1043 \times 58470$   |
|                                     | ais (4)             | $155 \times 2730$     |
|                                     | parity (30)         | $1044 \times 3614$    |

Table 2: Instance families used in our evaluations, grouped by domain. Variable (#Var) and clause (#Clause) counts are averages.

fix  $C_{\text{con\_dly}} = 200$ ,  $C_{\text{dec\_dly}} = 50$ ,  $C_{\text{con\_lit}} = 10$ ,  $C_{\text{con\_num}} = 10000$ , with  $C_{\text{uip}} = 20$  for Haplotype and  $C_{\text{uip}} = 140$  for SATLIB. No instance encountered failure except **pret**, where we retry once with  $C_{\text{wl}}$  increased by  $10\times$ ; we report the sum of both runs.

**Experiment design.** We start by evaluating the end-to-end performance of ppCDCL and then unpack *why* these gains occur via microbenchmarks and our parameter heuristics. Our experiments are designed to answer the following questions:

Q1. *Does ppCDCL outperform the prior privacy-preserving SAT solver ppSAT?*

In Section 6.1, we report the end-to-end performance of ppCDCL on a diverse set of CNF instances with varying sizes and degrees of hardness, and compare ppCDCL with ppSAT. We show that ppCDCL is both more capable and more efficient: it solves significantly more instances under a fixed timeout, and solves them much faster.

Q2. *How does ppCDCL scale with instance size?*

In Section 6.2, we show that ppCDCL has superior scalability: unlike ppSAT, ppCDCL avoids scans over the clause database, so CPS (cost per step) is nearly independent of the number of clauses. Moreover, thanks to our oblivious lazy data-structure design, the unit propagation cost is nearly constant.

Q3. *Do our parameter selection heuristics improve runtime and ensure low failure rates?*

In Section 6.3, we show that our parameter-selection heuristics produce tight bounds that minimize solving failures and excessive padding, resulting in significantly improved end-to-end runtime.

## 6.1 End-to-end Benchmarks

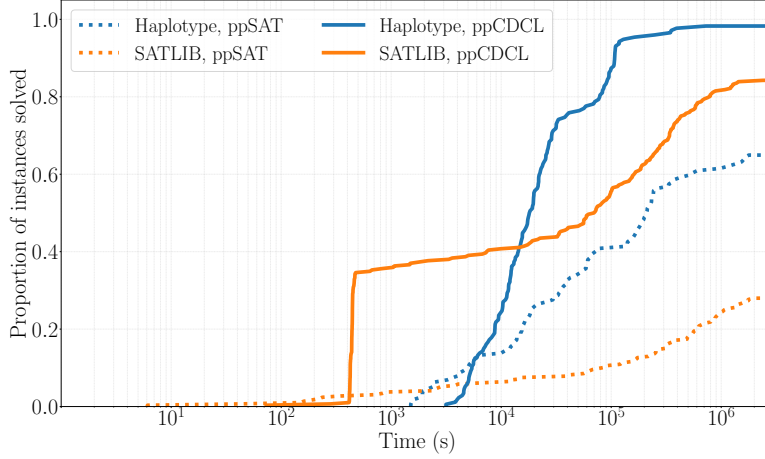


Figure 4: SATLIB and Haplotype benchmarks with the timeout set to  $3 \times 10^6$  s. ppCDCL solves more instances by the timeout, with 98.3% vs. 64.9% on easier Haplotype instances and 84.3% vs. 27.5% on harder SATLIB instances. ppCDCL is also more efficient on solved instances: within  $10^3$  seconds, ppCDCL solves 36.2% of SATLIB vs. 3.5% for ppSAT.

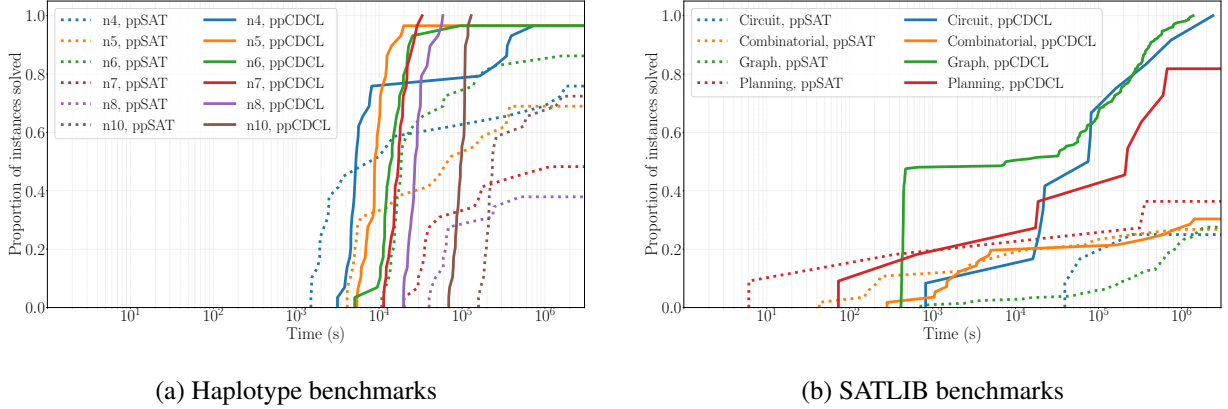


Figure 5: SATLIB and Haplotype benchmarks with the timeout set to  $3 \times 10^6$  s, grouped by instance domain. ppCDCL solves more instances on all subgroups and is significantly more efficient.

**Setup.** We measure the end-to-end runtime of ppCDCL on all of our instances (Table 2) and compare its runtime with that of ppSAT. For ppSAT, we follow the original paper [LJA<sup>+</sup>22] by (i) counting the number of steps  $S$  and (ii) computing the per-step 2PC cost  $t$ . The end-to-end cost is estimated as  $S \cdot t$ . To ensure a fair comparison, we re-evaluate ppSAT on our own testbed. For ppCDCL, we implement a plaintext version to obtain the number of invocations  $N_r$  for each subroutine  $r$ . We then compute the per-subroutine 2PC cost  $t_r$  and estimate runtime as  $\sum_r N_r \cdot t_r$ . To validate our estimation methodology, we execute ppCDCL end-to-end on formulas spanning different runtime scales. Across these instances, the measured runtime

and our projected estimate closely agree, with a mean relative error of 1.06%. We defer detailed results to Appendix A.1.

**Results.** Figure 5 and Figure 4 show the cumulative fraction of instances solved as a function of time, with a fixed timeout of  $3 \times 10^6$  seconds. Specifically, Figure 5a breaks down the Haplotype benchmarks by family, Figure 5b does the same for SATLIB, and Figure 4 aggregates all instances in Haplotype and SATLIB. We additionally compare ppCDCL and ppSAT with the plaintext SAT solver CaDiCaL [BFF<sup>+</sup>24] in Section 6.4. *Overall performance.* Figure 4 shows the overall results, aggregating all instances in Haplotype and SATLIB. The results show that ppCDCL consistently outperforms ppSAT in both efficiency and capability: it reaches higher solved fractions earlier and ultimately solves substantially more instances. At the timeout, ppCDCL solves **98.3%** of the Haplotype instances compared to **64.9%** for ppSAT. The gap is even wider on the more diverse SATLIB benchmarks, where ppCDCL solves **84.3%** of instances versus only **27.5%** for ppSAT. Notably, with a shorter timeout of  $10^3$  seconds, ppCDCL solves **36.2%** of the SATLIB instances, whereas ppSAT solves only **3.5%**.

*Detailed performance.* Figure 5 shows a detailed view of how ppCDCL performs on instances from different problem domains and with different sizes. Both Figure 5a and Figure 5b show that ppCDCL solves more instances at every Haplotype size and in every SATLIB problem domain. Notably, ppSAT fails to solve a single instance (**0%**) in the bf, flat200, and logistics benchmarks despite their moderate size. In contrast, ppCDCL solves **100%** of the bf and logistics instances and **96%** of the flat200 instances.

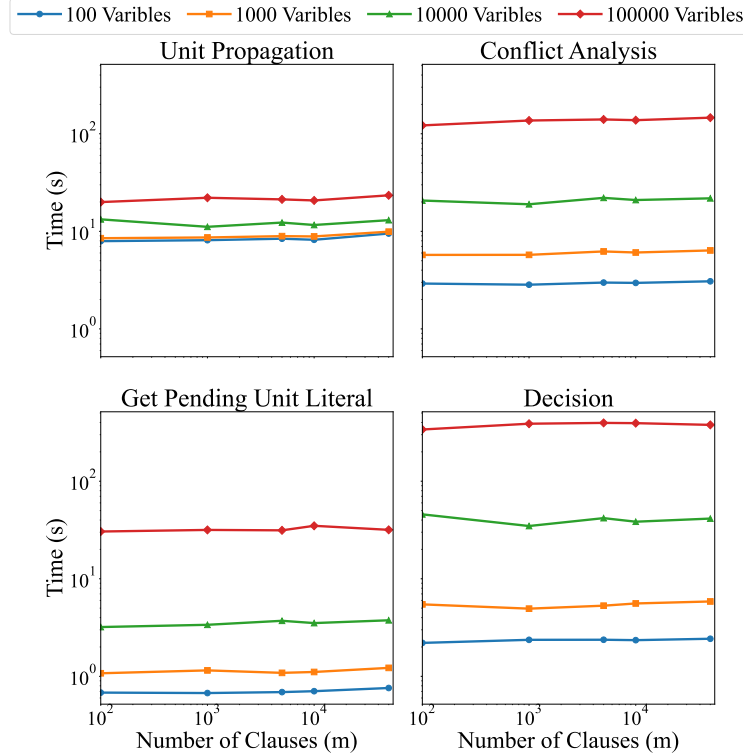


Figure 6: Runtime of ppCDCL’s procedures versus the number of clauses  $m$  and variables  $n$ . Costs are nearly constant in  $m$  because ppCDCL never scans the formula. The dominant growth with  $n$  comes from oblivious scans over variable-state arrays present in all procedures. The cost of the unit-propagation procedure remains lightweight for all  $m$  and  $n$  choices.

| Formula Size |         | Unit Propagation (s) |        | Decision (s) |        |
|--------------|---------|----------------------|--------|--------------|--------|
| #Var         | #Clause | ppSAT                | ppCDCL | ppSAT        | ppCDCL |
| 5K           | 10K     | 64.8                 | 11.1   | 266.4        | 26.2   |
| 10K          | 10K     | 127.3                | 12.3   | 532.6        | 42.0   |
| 10K          | 50K     | 638.5                | 13.9   | 2,663.8      | 42.7   |
| 50K          | 50K     | 3,206.5              | 20.2   | 13,292.1     | 206.1  |
| 50K          | 100K    | 6,412.9              | 18.8   | 26,646.2     | 208.1  |

Table 3: Cost of propagation and decision procedures in ppCDCL and ppSAT. The propagation cost for ppCDCL remains small and the decision cost scales primarily with #Var. In contrast, both costs in ppSAT inherently scale with  $\#Var \times \#Clause$ .

Overall, it is evident that ppCDCL consistently outperforms ppSAT on both the easier Haplotype benchmarks and the more complex SATLIB benchmarks. Our results indicate that ppCDCL preserves both the efficiency and effectiveness of CDCL in privacy-preserving settings, enabling significant progress towards practical solving on diverse, heterogeneous benchmarks where ppSAT fails.

## 6.2 Microbenchmarks

To understand the scalability of ppCDCL, we measure the wall-clock runtime of individual procedures in ppCDCL with varying  $n$  and  $m$ . Our results in Figure 6 highlight that the cost per step (CPS) is largely independent of the clause count  $m$  and is instead dominated by the number of variables  $n$ . When  $n$  is fixed, the AND-gate count remains constant, and the runtime fluctuates only slightly due to ORAM costs. In contrast, many subroutines of ppCDCL require linear-time operations on length- $n$  arrays. Conflict analysis is particularly expensive, as it requires multiple such scans. Propagation costs are largely independent of both  $m$  and  $n$ : ppCDCL scans a fixed number of literals per clause and performs a single ORAM range-read to fetch the watchlist. Our design improves propagation efficiency over ppSAT by a factor of 341 for large formulas with 50K variables and 100K clauses. We defer detailed discussion on complexity to Appendix B.2.

**Comparison with ppSAT.** We compare the microbenchmark results of ppCDCL and ppSAT for decision and propagation. Table 3 shows the results for varying formula sizes.

*Decision heuristics.* We compare the weighted randomized decision heuristic (wrand) in ppSAT with the VSIDS heuristic in ppCDCL. wrand is a simple heuristic that randomly selects a literal based on its occurrence in the formula; VSIDS is a more sophisticated conflict-driven heuristic widely used in modern solvers. As shown in Table 3, the decision cost of ppSAT scales with  $\#Var \times \#Clause$  as it scans the entire formula. In contrast, the decision cost of ppCDCL scales with only #Var and is lower by a factor of up to 128 for larger instances, despite using a more sophisticated heuristic.

*Unit propagation.* Table 3 compares the cost of one unit propagation step for both solvers. To propagate a unit literal  $\ell$ , ppSAT scans over the full formula/assignment matrix, and therefore the cost scales linearly with both #Var and #Clause; in contrast, ppCDCL visits only a single clause that is relevant to  $\ell$ , and therefore propagation costs remain small regardless of formula size.

## 6.3 Solver Heuristics and Parameter Choices

We now evaluate our oblivious clause learning heuristics and empirically validate our parameter choices.

| Inst. | Metric          | $C_{\text{con\_lit}}$ |             |             |             | Baseline |
|-------|-----------------|-----------------------|-------------|-------------|-------------|----------|
|       |                 | 5                     | 10          | 50          | 100         |          |
| n4    | <b>Time</b> (h) | <b>5.9</b>            | <b>23.2</b> | <b>37.0</b> | <b>61.9</b> | 1521.9   |
|       | <b>TO</b> (%)   | 20.7                  | 3.4         | 3.4         | 3.4         | 13.8     |
| n5    | <b>Time</b> (h) | <b>2.5</b>            | <b>2.7</b>  | <b>5.5</b>  | <b>9.1</b>  | 571.7    |
|       | <b>TO</b> (%)   | 3.4                   | 3.4         | 3.4         | 3.4         | 4.2      |
| n6    | <b>Time</b> (h) | <b>4.3</b>            | <b>4.9</b>  | <b>6.9</b>  | <b>11.4</b> | 631.4    |
|       | <b>TO</b> (%)   | 3.4                   | 3.4         | 3.4         | 3.4         | 3.4      |
| n7    | <b>Time</b> (h) | <b>5.1</b>            | <b>5.1</b>  | <b>11.3</b> | <b>18.5</b> | 1245.1   |
|       | <b>TO</b> (%)   | 0.0                   | 0.0         | 0.0         | 0.0         | 0.0      |
| n8    | <b>Time</b> (h) | <b>8.9</b>            | <b>8.5</b>  | <b>16.3</b> | <b>27.0</b> | 2410.1   |
|       | <b>TO</b> (%)   | 0.0                   | 0.0         | 0.0         | 0.0         | 0.0      |
| n10   | <b>Time</b> (h) | <b>27.9</b>           | <b>27.0</b> | <b>24.3</b> | <b>34.2</b> | 4102.8   |
|       | <b>TO</b> (%)   | 0.0                   | 0.0         | 0.0         | 0.0         | 0.0      |

Table 4: Effectiveness of our clause-learning scheme with varying  $C_{\text{con\_lit}}$ . **Baseline** implements clause-learning with worst-case padding. **TO** is the percentage of unsolved instances in  $3 \times 10^6$  seconds (we allow the baseline to run for an extended  $3 \times 10^7$  seconds). **Time** is the average solver runtime excluding timeout instances. Small  $C_{\text{con\_lit}}$  makes the solver discard low-quality learned clauses and significantly improves runtime in most cases.

**Effectiveness of clause learning parameters.** The clause-learning procedure is parameterized by  $C_{\text{uip}}$ ,  $C_{\text{con\_lit}}$ , and  $C_{\text{con\_num}}$ , which control the budget of our scheme. To demonstrate the efficiency of our design and how these parameters affect overall solver performance, we compare ppCDCL’s end-to-end runtime under different parameter values with that of a baseline solver that implements oblivious clause learning with worst-case padding ( $C_{\text{uip}} = C_{\text{con\_lit}} = \#\text{Var}$ ).

Table 4 reports ppCDCL runtime under different  $C_{\text{con\_lit}}$  values and a fixed  $C_{\text{uip}} = 20$ , with runtime entries averaged within each family excluding timeout instances. We allow the baseline to run with a longer timeout to demonstrate our improvements. For all instance families, our clause learning scheme outperforms the baseline by up to  $283\times$  in runtime. Moreover, a smaller  $C_{\text{con\_lit}}$  value improves runtime in most cases (e.g., 10 versus 100). For  $C_{\text{con\_num}}$ , we observe minimal impact on end-to-end runtime (Figure 8).

**Heuristics for selecting watchlist capacity parameter  $C_{\text{wl}}$ .** As mentioned in Section 5,  $C_{\text{wl}}$  is the most critical parameter as it affects both propagation efficiency and failure rate. Therefore, our heuristics for deriving  $C_{\text{wl}}$  must be safe and tight. Table 5 confirms that our global occurrence heuristic consistently yields  $C_{\text{wl}}$  values that are close to the minimum safe setting (small  $\Delta$ ) while remaining safe across all instances except **pret**. This exception matches the intuition from Section 5.2: these formulas have very few variables, so the resulting  $C_{\text{wl}}$  is numerically small, making it easier for even a modest number of learned clauses to exceed the bound. Table 5 reports, per family, the derived  $C_{\text{wl}}$  values from our global occurrence heuristic, which computes the maximum literal occurrence count in the joint formula  $\Phi_A \wedge \Phi_B$ .

**Effectiveness of scheduling parameters ( $C_{\text{dec\_dly}}$ ,  $C_{\text{con\_dly}}$ ).** We vary each scheduling delay independently and compare against a no-delay baseline ( $C_{\text{dec\_dly}} = C_{\text{con\_dly}} = 1$ ). Figure 7 shows that enabling a delay consistently improves runtime and prevents the corresponding procedure from dominating runtime. For

| Family    | Minimum | Heuristic |          |               |          |
|-----------|---------|-----------|----------|---------------|----------|
|           |         | $C_{wl}$  | $\Delta$ | Safe          | Time (s) |
| n4        | 26.9    | 31.6      | 17.4%    | <b>100.0%</b> | 0.03     |
| n5        | 27.0    | 32.4      | 20.0%    | <b>100.0%</b> | 0.04     |
| n6        | 29.0    | 34.1      | 18.2%    | <b>100.0%</b> | 0.06     |
| n7        | 29.4    | 33.1      | 12.8%    | <b>100.0%</b> | 0.08     |
| n8        | 32.3    | 34.1      | 5.4%     | <b>100.0%</b> | 0.10     |
| n10       | 40.0    | 40.2      | 0.5%     | <b>100.0%</b> | 0.17     |
| QG        | 220.80  | 340.40    | 54.2%    | <b>100.0%</b> | 0.46     |
| ais       | 21.00   | 21.33     | 1.6%     | <b>100.0%</b> | 0.01     |
| bf        | 18.22   | 32.33     | 77.4%    | <b>100.0%</b> | 0.21     |
| blocks    | 27.23   | 30.15     | 10.7%    | <b>100.0%</b> | 0.23     |
| flat100   | 10.00   | 10.00     | 0.0%     | <b>100.0%</b> | 0.02     |
| flat200   | 13.35   | 16.01     | 20.0%    | <b>100.0%</b> | 0.05     |
| logistics | 47.00   | 47.50     | 1.1%     | <b>100.0%</b> | 0.26     |
| ssa       | 96.72   | 97.67     | 1.0%     | <b>100.0%</b> | 0.31     |
| parity    | 23.22   | 23.35     | 0.6%     | <b>100.0%</b> | 0.11     |
| pret      | 46.50   | 4.83      | -89.6%   | 0.0%          | 0.01     |

Table 5: Runtime of our occurrence-based heuristic and derived  $C_{wl}$  values, averaged over each benchmark family. Timeout instances are excluded. **Minimum** is the minimum safe  $C_{wl}$  value observed by solving the instance with a plaintext CDCL solver. Using a  $C_{wl}$  smaller than the minimum will result in solver failure.  $\Delta$  is the relative gap to Minimum. **Safe** is the fraction of instances in the family where the heuristic-derived  $C_{wl}$  is at least Minimum. Our heuristic has negligible runtime and derives safe  $C_{wl}$  for the majority of the formulas.

the conflict delay  $C_{con\_dly}$ , larger values reduce the fraction of time spent in conflict analysis and improve end-to-end runtime (points shift toward the bottom-right). For the decision delay  $C_{dec\_dly}$ , enabling a delay is beneficial, but overly large values could hurt end-to-end runtime. Since decision steps are already inexpensive ( $< 2.5\%$  of total runtime), increasing  $C_{dec\_dly}$  yields little additional savings and can instead introduce idle time, leading to overall slowdowns when the x-axis value is below 1.

**Preprocessing effectiveness.** We evaluate our local preprocessing procedure and its impact on both ppCDCL and ppSAT (Appendix A.3). Preprocessing consistently reduces formula size, but its runtime improvements are reliable only for ppCDCL: it improves ppCDCL runtime for 20.7%–93.1% of instances and never increases the number of timeouts. In contrast, ppSAT improves on fewer instances and preprocessing even increases the number of timeouts on two families. These results suggest that preprocessing is beneficial when the solver can exploit the exposed structure; ppCDCL does so through propagation and conflict-driven learning, preserving CDCL’s algorithmic advantage over DPLL in the private setting.

## 6.4 Comparison with a Plaintext Solver

We compare the end-to-end runtimes of ppCDCL and ppSAT against CaDiCaL 2.1.3 [BFF<sup>+</sup>24], a state-of-the-art plaintext SAT solver. We evaluate CaDiCaL on all benchmark instances with internal preprocessing,

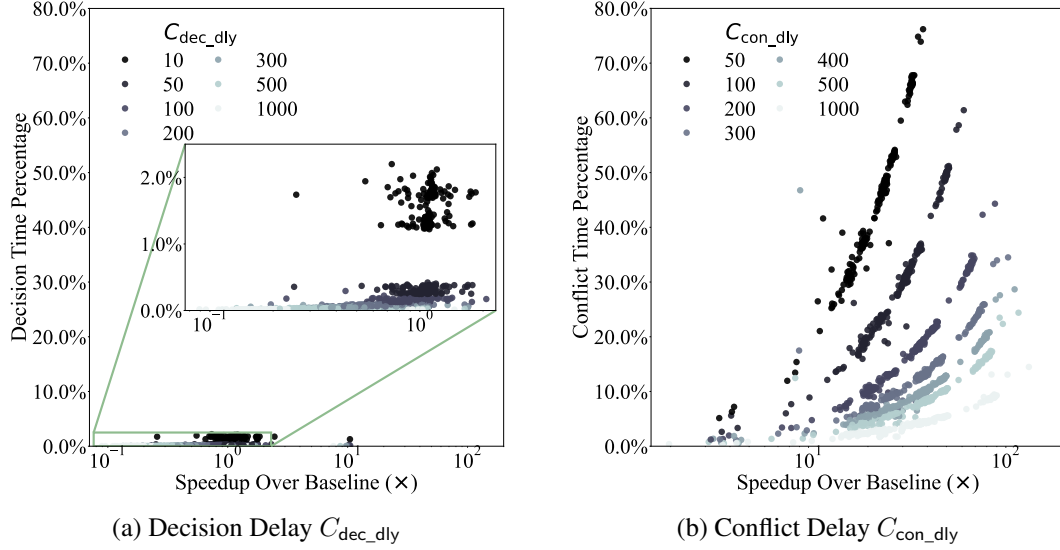


Figure 7: Effect of scheduling delays on runtime. Each point is one benchmark instance at a fixed delay value; darker colors indicate smaller delays (more frequent scheduling). The x-axis reports speedup over the no-delay baseline ( $> 1$  is faster), while the y-axis reports the fraction of total runtime spent in the scheduled subroutine. For decision delay  $C_{\text{dec\_dly}}$  (left), smaller delays improve runtime slightly, while overly large delays can cause slowdowns. For conflict delay  $C_{\text{con\_dly}}$  (right), larger delays consistently reduce conflict-analysis time and improve runtime.

| Domain           | # Instances | # Solved Instances |       |         | Time       |           |              | Overhead           |                    |
|------------------|-------------|--------------------|-------|---------|------------|-----------|--------------|--------------------|--------------------|
|                  |             | ppCDCL             | ppSAT | CaDiCaL | ppCDCL (s) | ppSAT (s) | CaDiCaL (ms) | ppCDCL             | ppSAT              |
| <i>Haplotype</i> | 174         | 171                | 113   | 174     | 16,794     | 43,406    | 5.22         | $3.22 \times 10^6$ | $8.32 \times 10^6$ |
| <i>SATLIB</i>    | 287         | 242                | 79    | 277     | 894        | 81,361    | 4.47         | $2.00 \times 10^5$ | $1.82 \times 10^7$ |

Table 6: Comparison between ppCDCL, ppSAT, and CaDiCaL on the Haplotype and SATLIB benchmarks. The timeout is  $3 \times 10^6$  s for ppCDCL and ppSAT and 1800 s for CaDiCaL. Times and overheads are geometric means over instances solved by all three solvers: 113 for Haplotype and 78 for SATLIB (191 overall). Overhead is the ppCDCL or ppSAT runtime divided by the CaDiCaL runtime on these instances.

no external preprocessing, and a 1800-second timeout<sup>5</sup>.

As Table 6 shows, ppCDCL significantly narrows the capability gap between privacy-preserving and plaintext SAT solvers. Across all benchmarks, ppCDCL solves 413 of 461 instances, approaching CaDiCaL’s 451 solved instances and more than doubling the 192 instances solved by ppSAT. On the larger and more diverse SATLIB benchmarks, ppCDCL solves about 3.1 times as many instances as ppSAT (242 versus 79), and is two orders of magnitude faster on instances solved by both. Nonetheless, despite these major algorithmic improvements over ppSAT, ppCDCL still incurs a substantial runtime overhead ( $\sim 10^6 \times$ ) relative to the highly optimized plaintext CaDiCaL solver.

<sup>5</sup>We note that a 1800-second plaintext timeout is sufficient for a fair comparison: instances requiring more than 1800 seconds to solve in CaDiCaL would far exceed the  $3 \times 10^6$  second timeout allowed for the two private solvers.

## Acknowledgements

Gefei Tan is supported by a Google PhD Fellowship. Xiao Wang is partially supported by NSF award CCF-2318975 and a Sloan Fellowship. Timos Antonopoulos and Ruzica Piskac are partially supported by NSF awards CCF-2219995, CNS-2245344, and CCF-2318974.

## References

- [AFT11] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *Journal of Artificial Intelligence Research*, 40:353–373, 2011.
- [AGCLS15] Carlos Ansótegui, Jesús Giráldez-Cru, Jordi Levy, and Laurent Simon. Using community structure to detect relevant learnt clauses. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 238–254. Springer, 2015.
- [AS09] Gilles Audemard and Laurent Simon. Glucose: a solver that predicts learnt clauses quality. *SAT Competition*, pages 7–8, 2009.
- [BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.
- [BF15] Armin Biere and Andreas Fröhlich. Evaluating cdcl restart schemes. *Proceedings of Pragmatics of SAT*, pages 1–17, 2015.
- [BFF<sup>+</sup>24] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleys, and Florian Pollitt. Cadical 2.0. In *International Conference on Computer Aided Verification*, pages 133–152. Springer, 2024.
- [BFW23] Armin Biere, Nils Froleys, and Wenxi Wang. Cadiback: Extracting backbones with cadical. In *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, 2023.
- [BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *ACM CCS 2016*. ACM Press, October 2016.
- [BJK21] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in sat solving. In *Handbook of Satisfiability*, pages 391–435. IOS press, 2021.
- [BN21] Sam Buss and Jakob Nordström. Proof complexity and sat solving. In *Handbook of Satisfiability*, pages 233–350. IOS Press, 2021.
- [Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY*, 13(1):143–202, 2000.
- [CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

- [Coo23] Stephen A Cook. The complexity of theorem-proving procedures. In *Logic, automata, and computational complexity: The works of Stephen A. Cook*, pages 143–152. Association for Computing Machinery, New York, NY, USA, 2023.
- [DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, July 1962.
- [DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, July 1960.
- [Ds17] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In *ACM CCS 2017*. ACM Press, October / November 2017.
- [FM09] John Franco and John Martin. A history of satisfiability. In *Handbook of satisfiability*, pages 3–74. IOS press, 2009.
- [GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT 2014*, volume 8441 of *LNCS*. Springer, Berlin, Heidelberg, May 2014.
- [GKK<sup>+</sup>12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *ACM CCS 2012*. ACM Press, October 2012.
- [GYW<sup>+</sup>23] Xiaojie Guo, Kang Yang, Xiao Wang, Wenhao Zhang, Xiang Xie, Jiang Zhang, and Zheli Liu. Half-tree: Halving the cost of tree expansion in COT and DPF. In *EUROCRYPT 2023, Part I*, volume 14004 of *LNCS*. Springer, Cham, April 2023.
- [HS00] Holger H Hoos and Thomas Stützle. Satlib: An online resource for research on sat. *Sat*, 2000:283–292, 2000.
- [Hua07] Jinbo Huang. A case for simple sat solvers. In *International Conference on Principles and Practice of Constraint Programming*, 2007.
- [KLW22] Tom Krüger, Jan-Hendrik Lorenz, and Florian Wörz. Too much information: Why cdcl solvers need to forget learned clauses. *Plos one*, 17(8):e0272967, 2022.
- [Kro09] Daniel Kroening. Software verification. In *Handbook of satisfiability*, pages 505–532. IOS Press, 2009.
- [KS99] Henry Kautz and Bart Selman. Unifying sat-based and graph-based planning. In *IJCAI*, volume 99, pages 318–325, 1999.
- [LGZ<sup>+</sup>15] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers. In *Haifa Verification Conference*, 2015.
- [LJA<sup>+</sup>22] Ning Luo, Samuel Judson, Timos Antonopoulos, Ruzica Piskac, and Xiao Wang. ppSAT: Towards two-party private SAT solving. In *USENIX Security 2022*. USENIX Association, August 2022.

- [LMS06a] Inês Lynce and João Marques-Silva. Efficient haplotype inference with boolean satisfiability. In *AAAI*, 2006.
- [LMS06b] Inês Lynce and João Marques-Silva. Sat in bioinformatics: Making the case with haplotype inference. In *Theory and Applications of Satisfiability Testing - SAT*, 2006.
- [LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.
- [MCP<sup>+</sup>06] Jonathan Marchini, David Cutler, Nick Patterson, Matthew Stephens, Eleazar Eskin, Eran Halperin, Shin Lin, Zhaohui S Qin, Heather M Munro, Gonçalo R Abecasis, and Peter Donnelly. A comparison of phasing algorithms for trios and unrelated individuals. *The American Journal of Human Genetics*, March 2006.
- [MMZ<sup>+</sup>01] Matthew W Moskwicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.
- [MSLM21] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, pages 133–182. IOS Press, Amsterdam, The Netherlands, 2021.
- [MSS99] João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.*, May 1999.
- [MSS00] João P. Marques-Silva and Karem A. Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of the 37th Annual Design Automation Conference (DAC)*, 2000.
- [PD07] Knot Pipatsrisawat and Adnan Darwiche. Rsat 2.0: Sat solver description. Technical report, Technical Report D–153, Automated Reasoning Group, Computer Science, 2007.
- [Rin21] Jussi Rintanen. Planning and sat. In *Handbook of Satisfiability: Second Edition*, pages 765–789. IOS Press, 2021.
- [SE05] Niklas Sorensson and Niklas Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005.
- [WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient multiparty computation toolkit. <https://github.com/emp-toolkit>, 2016.
- [ZGY<sup>+</sup>24] Wenhao Zhang, Xiaojie Guo, Kang Yang, Ruiyu Zhu, Yu Yu, and Xiao Wang. Efficient actively secure DPF and RAM-based 2PC with one-bit leakage. In *2024 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2024.
- [ZM88] Ramin Zabih and David A McAllester. A rearrangement search strategy for determining propositional satisfiability. In *AAAI*, volume 88, pages 155–160, 1988.
- [ZS00] Hantao Zhang and Mark Stickel. Implementing the davis–putnam method. *Journal of Automated Reasoning*, 2000.

## Open Science

Our documented prototype implementation is publicly available at:

<https://github.com/Gefei-Tan/ppCDCL-artifact>

## Ethical Considerations

Our work studies a privacy-preserving CDCL solver for two-party computation, whose intended benefit is to let mutually distrustful parties solve SAT instances without revealing their private inputs beyond the agreed public leakage. The main stakeholders are protocol users who contribute private formulas, organizations that may deploy such solvers, and end users affected by systems whose configuration or verification relies on SAT solving.

The primary benefit is improved privacy and efficiency: compared with prior privacy-preserving SAT approaches, ppCDCL brings modern SAT techniques such as clause learning and watched literals into 2PC while making data-dependent memory access and control flow oblivious. This can support privacy-preserving verification, configuration, planning, and analysis tasks where revealing either party’s constraints would be unacceptable. Potential risks include misuse of the protocol to perform private computation for harmful goals. We believe publication is justified because the work advances the public interest by improving privacy-preserving computation. The potential risks are not unique to our work and can be mitigated by existing legal and regulatory frameworks. As a result, we believe the benefits from publication outweigh the potential risks.

## A Additional Evaluations

### A.1 Validating End-to-End Cost Estimation

We validate the projection methodology from Section 6.1 by running ppCDCL end-to-end on representative CNF formulas and comparing the measured wall-clock runtime with the projected runtime. The selected formulas span different benchmark families and cover over two orders of magnitude in runtime. As shown in Table 7, the projection closely matches the measured execution time, with a mean relative error of 1.06%.

| Instance              | ppCDCL Runtime |              |       |
|-----------------------|----------------|--------------|-------|
|                       | Projected (s)  | Measured (s) | Error |
| uf20-01.cnf           | 21.3           | 21.6         | 1.39% |
| flat100/flat100-6.cnf | 252.9          | 254.1        | 0.47% |
| n4/genos.haps.2.cnf   | 2,334.3        | 2,303.7      | 1.33% |

Table 7: Projected and measured end-to-end runtime of ppCDCL. Runtimes are averaged over 5 runs on formulas spanning different benchmark families and scales. Error denotes the absolute relative difference.

### A.2 Evaluations on Parameter Choices

In this section, we provide additional results on our parameter-related experiments.

**Impact of parameters on runtime.** Figure 8 shows the impact of parameters  $C_{wl}$ ,  $C_{con\_lit}$ , and  $C_{con\_num}$  on solver runtime. Runtime scales linearly with  $C_{wl}$  as it determines the propagation overhead. A small  $C_{con\_lit}$

| Procedure                | Clause ORAM  |        | Variable State ORAM |        | Watchlist ORAM |        | Oblivious Linear Scans                         |
|--------------------------|--------------|--------|---------------------|--------|----------------|--------|------------------------------------------------|
|                          | R            | W      | R                   | W      | R              | W      |                                                |
| Get Pending Unit Literal | –            | –      | $O(n)$              | –      | $O(1)$         | –      | $O(n)$                                         |
| Decision                 | –            | –      | $O(n)$              | $O(1)$ | $O(1)$         | –      | $O(n)$                                         |
| Unit Propagation         | $O(1)$       | –      | $O(L)$              | $O(1)$ | $O(1)$         | $O(1)$ | $O(n + L + C_{wl})$                            |
| Conflict Analysis        | $O(C_{uip})$ | $O(1)$ | $O(C_{uip}L + n)$   | $O(n)$ | $O(1)$         | $O(1)$ | $O(C_{uip}(Ln + L^2 + n) + n^2 + nL + C_{wl})$ |

Table 8: Per-invocation complexity of ppCDCL procedures. We decouple the ORAM query complexity from the cost of oblivious linear scans. Here,  $L = \max(C_{phi\_lit}, C_{con\_lit})$ , and **R/W** denote read/write operations. A Clause ORAM read requires querying both the original-clause and learned-clause ORAMs, whereas a write requires a single query to the learned-clause ORAM. The  $O(\cdot)$  notation hides constants independent of  $n$ ,  $m$ ,  $L$ ,  $C_{uip}$ ,  $C_{wl}$ , and  $C_{con\_num}$ .

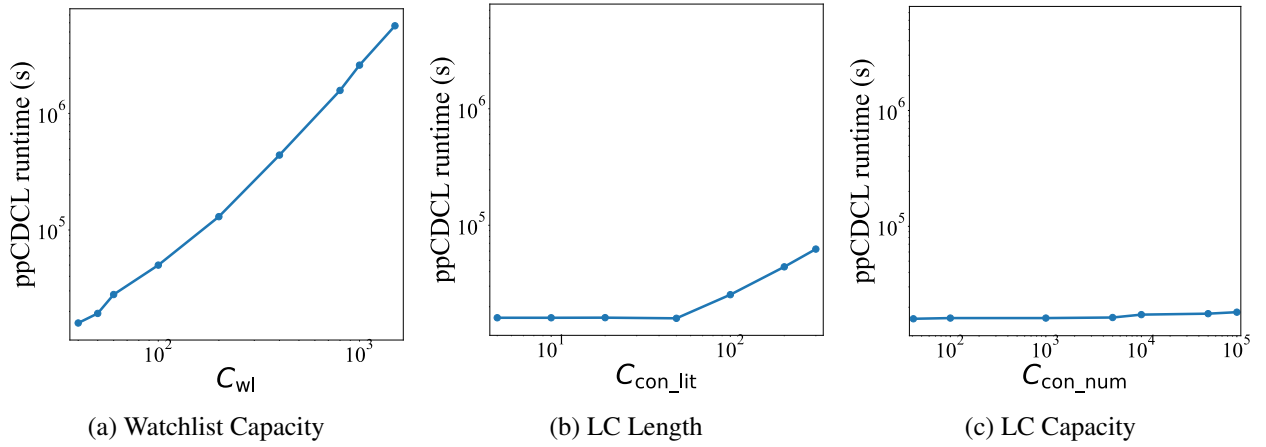


Figure 8: **Impact of public parameters on ppCDCL runtime.**  $C_{wl}$  sets the watchlist capacity,  $C_{con\_lit}$  sets the maximum length of a learned clause (LC), and  $C_{con\_num}$  is the capacity of the learned-clause ORAM. Numbers reported are from solving the instance `n4-genos.haps.4.cnf`, but the cost model is independent of specific instances.  $C_{wl}$  has the largest effect on runtime, as it sets the number of steps to propagate a single literal. A small  $C_{con\_lit}$  improves solver runtime compared with a larger  $C_{con\_lit}$  by discarding lengthy learned clauses.  $C_{con\_num}$  has minimal effect on solver runtime.

results in a small runtime, while an unnecessarily large  $C_{con\_lit}$  causes the runtime to scale linearly.  $C_{con\_num}$  has minimal impact on solver runtime.

### A.3 Evaluating Local Preprocessing

**Preprocessing in SAT solving.** Preprocessing [BJK21] is a standard component of modern SAT solving: a formula is first simplified so that the subsequent solver search is performed on a smaller or more structured instance. In our setting, preprocessing is performed locally by each party before the online ppCDCL execution.

Algorithm 2 describes our preprocessing procedure. Given local formulas  $\Phi_A$  and  $\Phi_B$ , each party first extracts backbone literals from its own formula and treats them as unit clauses. It then runs a local CDCL solver for a bounded number of steps to collect learned clauses, optionally filtering them by length and

---

**Algorithm 2:** Preprocessing

---

**Input:** local CNF formulas  $\Phi_A, \Phi_B$ , preprocessing parameters  $max\_steps, len\_cap$ , and  $lbd\_cap$

**Output:** UNSAT, or preprocessed CNFs  $\Phi'_A, \Phi'_B$  and solver-parameter tuple

$$\theta = (C_{wl}, C_{con\_lit}, C_{con\_num}, C_{uip}, C_{con\_dly}, C_{dec\_dly})$$

**for**  $i \in \{A, B\}$  **do**

$B_i \leftarrow \text{ExtractBackbone}(\Phi_i)$

**if**  $B_i = \perp$  **then**

$\Phi'_i \leftarrow \Phi_i$

**else if**  $B_i = \text{UNSAT}$  **then**

**return** *UNSAT*

**else**

        // local CDCL solving

$C_i \leftarrow \text{CDCL}(\Phi_i, max\_steps)$

$C'_i \leftarrow \text{FilterLearned}(C_i, len\_cap, lbd\_cap)$

$\Phi'_i \leftarrow \text{simplify}(\Phi_i \wedge B_i \wedge C'_i)$

$\theta \leftarrow \text{DeriveParameter}(\Phi'_A, \Phi'_B)$

**return**  $(\Phi'_A, \Phi'_B, \theta)$ 

---

LBD score [AS09]. The retained clauses are appended to the original formula, and the resulting formula is simplified. The preprocessed formulas determine  $C_{wl}$  through the global-occurrence heuristic; together with the fixed public choices of  $C_{con\_lit}$ ,  $C_{con\_num}$ ,  $C_{uip}$ ,  $C_{con\_dly}$ , and  $C_{dec\_dly}$ , this yields the solver-parameter tuple  $\theta$  used by ppCDCL. Combined with the agreed public CNF bounds  $(n, m, C_{phi\_lit})$  and step budget  $max\_t$ ,  $\theta$  forms the online public configuration  $p$  defined in Figure 3.

**Preprocessing effectiveness.** We evaluate preprocessing along two dimensions: formula-size reduction and online runtime improvement. To separate the effect of preprocessing from the effect of the solver design, we apply the same preprocessing pipeline to both ppCDCL and ppSAT.

Table 9 shows that preprocessing consistently reduces the number of clauses. However, the resulting runtime impact differs substantially between the two solvers. For ppCDCL, preprocessing often improves runtime: the simplified formula exposes structure that can be exploited by unit propagation, conflict analysis, and learned clauses. For ppSAT, the gains are less consistent, and preprocessing can even increase the number of timeouts. This is because reducing formula size does not necessarily make the remaining search easier for a DPLL-style solver; the simplified formula may concentrate the instance into a smaller but more constrained core.

These results show that preprocessing is not automatically a free speedup. Its benefit depends on whether the solver can exploit the structural information exposed during simplification. In our experiments, ppCDCL benefits more reliably from this structure than ppSAT.

## B Further Discussion

### B.1 Restarts in Bounded Clause Learning

When conflict analysis exceeds  $C_{uip}$ , the learned clause exceeds  $C_{con\_lit}$ , or the public learned-clause append region is exhausted, ppCDCL discards the current conflict-analysis result and restarts. A well-known theoretical limitation of restarts is the potential loss of solver completeness. If restarts repeatedly interrupt the

| Family | #Clause ↓ | ppSAT   |          | ppCDCL |          |
|--------|-----------|---------|----------|--------|----------|
|        |           | TO      | Improved | TO     | Improved |
| n4     | 29.7%     | 7 → 10  | 37.9%    | 3 → 1  | 79.3%    |
| n5     | 29.2%     | 9 → 10  | 51.7%    | 1 → 1  | 93.1%    |
| n6     | 28.3%     | 4 → 4   | 65.5%    | 1 → 1  | 86.2%    |
| n7     | 28.6%     | 14 → 12 | 44.4%    | 0 → 0  | 81.5%    |
| n10    | 28.3%     | 8 → 6   | 75.9%    | 0 → 0  | 20.7%    |

Table 9: **Impact of preprocessing on ppSAT and ppCDCL performance.** “→” denotes the change from before to after preprocessing. We report only clause reduction (#Clause ↓) because preprocessing does not change the number of variables in these instances. **TO** is the number of instances not solved within  $3 \times 10^6$  seconds. **Improved** is the fraction of instances whose runtime decreases after preprocessing. Preprocessing is effective for ppCDCL, improving runtime for up to 93% of the instances. In contrast, improvements for ppSAT are far less consistent, and preprocessing can significantly degrade ppSAT performance, causing more instances to time out. ppSAT’s DPLL search benefits less consistently from this preprocessing.

search before enough systematic progress is made, the solver risks getting trapped in an infinite loop. In the standard SAT literature, completeness is recovered by enforcing an unboundedly increasing restart interval, such as the Luby sequence adopted in ppCDCL. In principle, if processing every useful conflict hits a cap, the solver could restart indefinitely without monotonically exhausting the search tree. In ppCDCL, such a case leads to TIMEOUT under the public step budget.

Completeness can be achieved by placing chronological backtracking between restarts: instead of restarting on every cap hit, the solver can be configured to allow a monotonically increasing number of chronological backtracks between restarts. In practice, we adopt restarts in our implementation because they outperform chronological backtracking even under aggressively small budgets.

## B.2 Complexity

To analyze the complexity of ppCDCL procedures, we separate the ORAM accesses from the remaining oblivious linear scans. We evaluate and report the access pattern in terms of abstract ORAM operations, as this isolates our algorithmic complexity from the concrete performance of the underlying ORAM backend. Table 8 summarizes the asymptotic number of ORAM queries alongside the complexity of the remaining oblivious linear scans for each ppCDCL procedure.

## C Algorithms for ppCDCL

We provide the top-level ppCDCL algorithm (Algorithm 3) and the subprocedures it invokes.

For simplicity, we assume all procedures have access to all data structures and variables as initialized in the main ppCDCL algorithm. We use  $\text{lf}(b, x, y)$  to represent branching on secret values, i.e.,  $\text{lf}(b, x, y) = y(1 - b) + bx$ . We slightly abuse the notation to allow  $x$  and  $y$  to be function calls (e.g., read/write calls to data structures). We abstract some conditional branching away for simplicity and write all data-structure accesses using  $d \leftarrow D[i]$ , while their underlying implementations are all oblivious. Similarly, we use expressions such as  $A.\text{remove}(i, b)$  and  $\text{Set}.\text{findMaxVal}(b)$  as shorthands for oblivious subroutines implemented by linear scans. For readability, we present the algorithms using high-level data-structure

operations, treating some oblivious subroutines as shorthands. Each such operation is implemented with the appropriate dummy accesses, linear scans, or ORAM reads/writes, and its cost is included in our analysis. For example,  $V.\text{setDecision}(\ell, V.\text{cur\_dl}, b_{\text{dec}})$  conditionally records both the assignment and decision level of  $\ell$  in the variable-state ORAM. Similarly,  $\text{CheckSAT}(b_{\text{act}})$  scans the variable-state ORAM and returns 1 iff  $b_{\text{act}} = 1$ , all variables are assigned, and  $U$  contains no pending unit literal.

Reads at the dummy clause ID, literal key, or variable index  $\perp$  are designated dummy accesses that return all- $\perp$  padded values. A data-structure operation with a false active flag leaves all real state unchanged. The learned-clause ORAM is initialized to all- $\perp$  entries; a false-active scheduled write still performs the physical access to its public slot but leaves that dummy entry unchanged.  $WL.\text{init}$  and  $WL.\text{write\_clause}$  create reciprocal watch records: if clause  $c$  watches  $(a, b)$ , then  $WL[a]$  stores  $(c, b)$  and  $WL[b]$  stores  $(c, a)$ . The shorthand  $(A, b_{\text{cap}}) \leftarrow \text{AddLitBounded}(A, \ell, b, b_{\text{cap}})$  performs a fixed scan of the  $C_{\text{con\_lit}}$  slots in  $A$ , conditionally inserts a non-dummy  $\ell$  into the first empty slot, and sets the sticky flag  $b_{\text{cap}}$  if an enabled insertion finds no empty slot.

---

**Algorithm 3: Privacy-Preserving CDCL Solver (ppCDCL)**

---

**Input:** CNF formula  $\Phi = \Phi_A \wedge \Phi_B$ , public configuration  $p = (n, m, C_{\text{phi\_lit}}, \theta, \text{max\_t})$ , where  $\theta = (C_{\text{wl}}, C_{\text{con\_lit}}, C_{\text{con\_num}}, C_{\text{uiip}}, C_{\text{con\_dly}}, C_{\text{dec\_dly}})$

**Output:** (SAT,  $\alpha \models \Phi$ ) / UNSAT / TIMEOUT / FAILURE

**Initialize:**  $t \leftarrow 0$ ;  $w_{\text{ord}} \leftarrow \lceil \log_2(n + 3\text{max\_t} + 1) \rceil$ ;  $\text{ord} : \text{uint}_{w_{\text{ord}}} \leftarrow n$ ;  $\ell \leftarrow \perp$ ;  $c \leftarrow \perp$ ;  $\text{dec\_pend} \leftarrow 0$ ;  $\text{con\_pend} \leftarrow 0$ ;  $\text{conf\_idx} \leftarrow \perp$ ;  $\text{learn\_slot} \leftarrow 1$ ;  $\text{curwl} \leftarrow \perp$ ;  $\text{U.init}(n)$ ;  $b_{\text{fail}} \leftarrow \text{WL.init}(\Phi, n, C_{\text{wl}})$ ;  $\text{V.init}(n, m, C_{\text{con\_num}})$ ;  $\text{V.order}[1..n] : \text{uint}_{w_{\text{ord}}}^n \leftarrow [1, \dots, n]$ ;  $\text{Score.init}(n, C_{\text{con\_num}})$ ;

**if**  $b_{\text{fail}}$  **then return FAILURE**

**if**  $\text{CheckUnitContradiction}(\Phi)$  **then return UNSAT**

**while**  $t < \text{max\_t}$  **do**

$t \leftarrow t + 1$

// public schedule for this step

$b_{\text{dec}}, b_{\text{con}}, b_u, \text{ctr} \leftarrow \text{Schedule}(t, C_{\text{wl}}, C_{\text{dec\_dly}}, C_{\text{con\_dly}})$

// do dummy work if conf/dec is pending

$\text{is\_idle} \leftarrow \text{con\_pend} \vee \text{dec\_pend}$

**if**  $b_u$  **then**

// refresh wl from pending unit literals

$\ell_u \leftarrow \text{GetPendingUnitLiteral}(\neg \text{is\_idle})$

$\text{hasU} \leftarrow (\ell_u \neq \perp)$

$\text{takeU} \leftarrow (\neg \text{is\_idle}) \wedge \text{hasU}$

$\ell \leftarrow \text{If}(\text{takeU}, \ell_u, \ell)$

$\text{curwl} \leftarrow \text{If}(\text{takeU}, \text{WL.get}(-\ell_u), \text{curwl})$

$b_{\text{chk}} \leftarrow (\neg \text{is\_idle}) \wedge (\neg \text{hasU})$

$b_{\text{sat}} \leftarrow \text{CheckSAT}(b_{\text{chk}})$

**if**  $b_{\text{sat}}$  **then**

// Reveal the satisfying assignment

**return** {SAT,  $\text{V.reveal}()$ }

$\text{dec\_pend} \leftarrow \text{If}(b_{\text{chk}} \wedge (\neg b_{\text{sat}}), 1, \text{dec\_pend})$

**if**  $b_{\text{dec}}$  **then**

// refresh wl from new decision

$\ell_d \leftarrow \text{Decision}(\text{dec\_pend})$

$\ell \leftarrow \text{If}(\text{dec\_pend}, \ell_d, \ell)$

$\text{curwl} \leftarrow \text{If}(\text{dec\_pend}, \text{WL.get}(-\ell_d), \text{curwl})$

$\text{dec\_pend} \leftarrow 0$

// continue scanning current wl

$\text{is\_idle} \leftarrow \text{con\_pend} \vee \text{dec\_pend}$

$b_{\text{cur\_wl}} \leftarrow \neg \text{is\_idle} \wedge (\ell \neq \perp)$

$(c, \ell') \leftarrow \text{If}(b_{\text{cur\_wl}}, \text{curwl}[\text{ctr}], (\perp, \perp))$

$\ell_{\text{old}} \leftarrow \text{If}(b_{\text{cur\_wl}}, -\ell, \perp)$

$(\text{con\_pend\_t}, b_{\text{fail}}) \leftarrow \text{UnitPropagation}(c, \ell_{\text{old}}, \ell')$

**if**  $b_{\text{fail}}$  **then**

**return FAILURE** // watchlist overflow

// cache current clause if conflict occurs

$\text{con\_pend} \leftarrow \text{con\_pend} \vee \text{con\_pend\_t}$

$\text{conf\_idx} \leftarrow \text{If}(\text{con\_pend\_t}, c, \text{conf\_idx})$

**if**  $b_{\text{con}}$  **then**

// handle a conflict every  $C_{\text{con\_dly}}$  steps

$(b_{\text{unsat}}, b_{\text{fail}}) \leftarrow \text{Conflict}(\text{con\_pend}, \text{conf\_idx}, \text{learn\_slot})$

$\text{learn\_slot} \leftarrow \min(\text{learn\_slot} + 1, C_{\text{con\_num}} + 1)$

$\text{con\_pend} \leftarrow 0$ ;  $\text{conf\_idx} \leftarrow \perp$

**if**  $b_{\text{fail}}$  **then return FAILURE**

**if**  $b_{\text{unsat}}$  **then return UNSAT**

**return TIMEOUT**

---

---

**Algorithm 4:** Decision

---

**Input:** active flag  $b_{\text{dec}}$   
**Output:**  $\ell_d$   
**Initialize:**  $best\_score \leftarrow 0$ ;  $best\_var \leftarrow 1$ ;  $best\_phase \leftarrow 1$ ;  
 $V.cur\_dl \leftarrow \text{If}(b_{\text{dec}}, V.cur\_dl + 1, V.cur\_dl)$   
**for**  $i \leftarrow 1$  **to**  $n$  **do**  
     $u \leftarrow V.\text{isUnassigned}(i)$   
     $pick \leftarrow b_{\text{dec}} \wedge u \wedge (\text{Score}[i] \geq best\_score)$   
     $best\_score \leftarrow \text{If}(pick, \text{Score}[i], best\_score)$   
     $best\_var \leftarrow \text{If}(pick, i, best\_var)$   
     $best\_phase \leftarrow \text{If}(pick, V.\text{phase}[i], best\_phase)$   
 $pos \leftarrow (best\_phase = 1)$   
 $\ell_d \leftarrow \text{If}(pos, best\_var, -best\_var)$   
 $V.\text{setDecision}(\ell_d, V.cur\_dl, b_{\text{dec}})$   
 $ord \leftarrow ord + b_{\text{dec}}$   
 $V.\text{order}[|\ell_d|] \leftarrow \text{If}(b_{\text{dec}}, ord, V.\text{order}[|\ell_d|])$   
**return**  $\ell_d$

---

---

**Algorithm 5:** CheckClause

---

**Input:** clause  $c$ , falsified watch  $\ell_{old}$ , kept watch  $\ell'$ , active flag  $b_{act}$   
**Output:**  $\{b_{conf}, b_{unit}, b_{wl}, x, y\}$   
**Initialize:**  $x \leftarrow \ell'$ ;  $y \leftarrow \ell'$ ;  $b_{rep} \leftarrow 0$ ;  
**for**  $j \leftarrow 1$  **to**  $\max(C_{\text{con\_lit}}, C_{\text{phi\_lit}})$  **do**  
     $\lambda \leftarrow c[j]$   
     $b_{cand} \leftarrow b_{act} \wedge (\lambda \neq \perp) \wedge (\lambda \neq \ell_{old}) \wedge (\lambda \neq \ell') \wedge \neg V.\text{isFalsified}(\lambda)$   
     $y \leftarrow \text{If}(b_{cand} \wedge (\neg b_{rep}), \lambda, y)$   
     $b_{rep} \leftarrow b_{rep} \vee b_{cand}$   
 $b_{wl} \leftarrow b_{act} \wedge b_{rep}$   
 $b_{conf} \leftarrow b_{act} \wedge (\neg b_{rep}) \wedge (\ell' \neq \perp) \wedge V.\text{isFalsified}(\ell')$   
 $b_{unit} \leftarrow b_{act} \wedge (\neg b_{rep}) \wedge (\ell' \neq \perp) \wedge V.\text{isUnassigned}(|\ell'|)$   
// conflict if  $x = y = \perp$ ; unit if  $x = y = \ell'$ ; update WL if  $x = \ell'$ ,  $y \neq x$   
 $x \leftarrow \text{If}(b_{conf}, \perp, x)$   
 $y \leftarrow \text{If}(b_{conf}, \perp, y)$   
**return**  $\{b_{conf}, b_{unit}, b_{wl}, x, y\}$

---

---

**Algorithm 6:** Schedule

---

**Input:** current step  $t$ , parameters  $\{C_{wl}, C_{dec\_dly}, C_{con\_dly}\}$   
**Output:**  $\{b_{dec}, b_{con}, b_u, ctr\}$   
**Initialize:**  $b_{dec} \leftarrow 0; b_{con} \leftarrow 0; b_u \leftarrow 0; ctr \leftarrow 0;$   
 $b_u \leftarrow (t \bmod C_{wl} = 0)$  // unit-prop microstep  
 $b_{dec} \leftarrow (t \bmod C_{dec\_dly} = 0)$  // decision step  
 $b_{con} \leftarrow (t \bmod C_{con\_dly} = 0)$  // conflict-analysis step  
 $ctr \leftarrow (t \bmod C_{wl})$   
 $ctr \leftarrow \text{If}(ctr = 0, C_{wl}, ctr)$   
**return**  $\{b_{dec}, b_{con}, b_u, ctr\}$

---

---

**Algorithm 7:** UnitPropagation

---

**Input:** clause ID  $c_{idx}$ , falsified watch  $\ell_{old}$ , kept watch  $\ell'$   
**Output:** conflict flag  $b_{conf}$ , watchlist-overflow flag  $b_{fail}$   
**Initialize:**  $b_{act} \leftarrow (c_{idx} \neq \perp); b_{conf} \leftarrow 0; x \leftarrow 0; y \leftarrow 0;$   
 $c \leftarrow \text{CL}[c_{idx}]$   
 $\{b_{conf}, b_{unit}, b_{wl}, x, y\} \leftarrow \text{CheckClause}(c, \ell_{old}, \ell', b_{act})$   
 $\text{AddImplication}(x, c_{idx}, b_{unit})$   
 $b_{fail} \leftarrow \text{UpdateWL}(c_{idx}, \ell_{old}, x, y, b_{wl})$   
**return**  $b_{conf}, b_{fail}$

---

---

**Algorithm 8:** AddImplication

---

**Input:** implied literal  $x$ , antecedent clause ID  $c_{idx}$ , active flag  $b_{act}$   
**Output:**  
 $ass \leftarrow \text{If}((x > 0), 1, 0)$  // update variable state  
 $u_x \leftarrow \text{If}((x > 0), 2, 1)$   
 $V.ass[x] \leftarrow \text{If}(b_{act}, ass, V.ass[x])$   
 $V.dl[x] \leftarrow \text{If}(b_{act}, V.cur\_dl, V.dl[x])$   
 $V.ante[x] \leftarrow \text{If}(b_{act}, c_{idx}, V.ante[x])$   
 $ord \leftarrow ord + b_{act}$   
 $V.order[x] \leftarrow \text{If}(b_{act}, ord, V.order[x])$   
**for**  $i \leftarrow 1$  **to**  $n$  **do**  
     $U[i] \leftarrow \text{If}(b_{act} \wedge (|x| = i), u_x, U[i])$  // enqueue

---

---

**Algorithm 9: GetPendingUnitLiteral**

---

**Input:** active flag  $b_{act}$   
**Output:** unit literal  $\ell_u$   
**Initialize:**  $\ell_u \leftarrow \perp$ ;  $found \leftarrow 0$ ;  
**for**  $i \leftarrow 1$  **to**  $n$  **//** linear scan to find the first pending literal  
**do**  
     $val \leftarrow U[i]$   
     $is\_pending \leftarrow (val \neq 0)$   
     $pick \leftarrow b_{act} \wedge is\_pending \wedge \neg found$   
     $lit \leftarrow \text{If}(val = 2, i, -i)$  **//** decode  $1 \mapsto -i$  and  $2 \mapsto i$   
     $\ell_u \leftarrow \text{If}(pick, lit, \ell_u)$   
     $U[i] \leftarrow \text{If}(pick, 0, val)$  **//** clear pending status  
     $found \leftarrow found \vee pick$   
**return**  $\ell_u$

---

---

**Algorithm 10: UpdateWL**

---

**Input:** clause ID  $c_{idx}$ , old watch  $\ell_{old}$ , kept watch  $x$ , replacement watch  $y$ , active flag  $b_{act}$   
**Output:** watchlist-overflow flag  $b_{fail}$   
**//** each helper performs a fixed scan of one padded watchlist  
 $WL[\ell_{old}].\text{remove}((c_{idx}, x), b_{act})$   
 $WL[x].\text{replace}((c_{idx}, \ell_{old}), (c_{idx}, y), b_{act})$   
 $b_{insfail} \leftarrow WL[y].\text{insert}((c_{idx}, x), b_{act})$   
 $b_{fail} \leftarrow b_{act} \wedge b_{insfail}$   
**return**  $b_{fail}$

---

---

**Algorithm 11: Backtrack**

---

**Input:** level  $\beta$ , active flag  $b_{act}$   
**Output:**  
 $V.cur\_dl \leftarrow \text{If}(b_{act}, \beta, V.cur\_dl)$   
**//** revert assignments  
**for**  $i \leftarrow 1$  **to**  $n$  **do**  
     $d \leftarrow V.dl[i]; ass \leftarrow V.ass[i]; b_{rst} \leftarrow b_{act} \wedge (d > \beta)$   
     $V.ass[i] \leftarrow \text{If}(b_{rst}, \perp, ass)$   
     $V.dl[i] \leftarrow \text{If}(b_{rst}, \perp, V.dl[i])$   
     $V.ante[i] \leftarrow \text{If}(b_{rst}, \perp, V.ante[i])$   
     $U[i] \leftarrow \text{If}(b_{act}, 0, U[i])$   
     $b_{ucl} \leftarrow b_{act} \wedge (d = 0) \wedge (ass \neq \perp)$  **//** unit clause  
     $u_i \leftarrow \text{If}(ass = 1, 2, 1)$   
     $U[i] \leftarrow \text{If}(b_{ucl}, u_i, U[i])$  **//** push unit clause

---

---

**Algorithm 12: Conflict**

---

**Input:** active flag  $b_{act}$ , conflicting clause ID  $conf\_idx$ , public learned slot  $learn\_slot$   
**Output:** UNSAT flag  $b_{unsat}$ , watchlist-overflow flag  $b_{fail}$   
**Initialize:**  $L \leftarrow \max(C_{con\_lit}, C_{phi\_lit})$   
 $c_{idx} \leftarrow conf\_idx$ ;  $pivot \leftarrow 0$ ;  $pathC \leftarrow 0$ ;  $b_{uipcap} \leftarrow 0$ ;  $b_{litcap} \leftarrow 0$   
 $OutCl \leftarrow [\perp]^{C_{con\_lit}}$ ;  $CandVar \leftarrow \emptyset$ ;  $CandOrd \leftarrow \emptyset$ ;  $Touched \leftarrow \emptyset$   
 $b_{unsat} \leftarrow b_{act} \wedge (V.cur\_dl = 0)$   
 $b_{work} \leftarrow b_{act} \wedge \neg b_{unsat}$ ;  $b_{do} \leftarrow b_{work}$   
**for**  $r \leftarrow 1$  **to**  $C_{uip}$  **do**  
     $c \leftarrow CL[c_{idx}]$   
    // scan  $c$ : current-level newly seen literals go to candidates; other  
    // newly seen non-root literals go to  $OutCl$   
    **for**  $j \leftarrow 1$  **to**  $L$  **do**  
         $\ell \leftarrow c[j]$ ;  $v \leftarrow |\ell|$ ;  $lvl \leftarrow V.dl[v]$   
         $b_{new} \leftarrow b_{do} \wedge (\ell \neq \perp) \wedge (v \neq |pivot|) \wedge (lvl \neq 0) \wedge \neg Seen[v]$   
         $b_{cur} \leftarrow b_{new} \wedge (lvl = V.cur\_dl)$   
         $b_{old} \leftarrow b_{new} \wedge \neg b_{cur}$   
         $Seen[v] \leftarrow \text{If}(b_{new}, 1, Seen[v])$   
         $Touched.add(v, b_{new})$  // linear scan to find an empty slot  
         $CandVar.add(v, b_{cur})$ ;  $CandOrd.add(V.order[v], b_{cur})$   
         $pathC \leftarrow \text{If}(b_{cur}, pathC + 1, pathC)$   
         $(OutCl, b_{litcap}) \leftarrow \text{AddLitBounded}(OutCl, \ell, b_{old}, b_{litcap})$   
    // pick most recent current-level literal  
     $idx \leftarrow CandOrd.max\_idx(b_{do})$  // linear scan  
     $pvar \leftarrow CandVar[idx]$   
     $asg \leftarrow V.ass[pvar]$   
     $pivot' \leftarrow \text{If}(asg = 1, pvar, -pvar)$   
     $pivot \leftarrow \text{If}(b_{do}, pivot', pivot)$   
     $CandVar.remove(idx, b_{do})$ ;  $CandOrd.remove(idx, b_{do})$  // linear scan  
     $Seen[|pivot|] \leftarrow \text{If}(b_{do}, 0, Seen[|pivot|])$   
     $pathC \leftarrow \text{If}(b_{do}, pathC - 1, pathC)$   
     $b_{stop} \leftarrow b_{do} \wedge (pathC = 0)$   
     $b_{uipcap} \leftarrow b_{do} \wedge (r = C_{uip}) \wedge \neg b_{stop}$   
     $next\_idx \leftarrow V.ante[|pivot|]$   
     $c_{idx} \leftarrow \text{If}(b_{stop} \vee b_{uipcap}, c_{idx}, next\_idx)$   
     $b_{do} \leftarrow b_{do} \wedge \neg b_{stop} \wedge \neg b_{uipcap}$   
 $\alpha \leftarrow -pivot$   
 $b_{complete} \leftarrow b_{work} \wedge \neg b_{uipcap}$   
 $(OutCl, b_{litcap}) \leftarrow \text{AddLitBounded}(OutCl, \alpha, b_{complete}, b_{litcap})$   
 $\beta \leftarrow \text{MaxLvlBelowCur}(OutCl)$  // linear scan  
 $b_{slot} \leftarrow (learn\_slot \leq C_{con\_num})$ ;  $b_{numcap} \leftarrow b_{work} \wedge \neg b_{slot}$   
// save the clause if no cap is hit  
 $b_{\ell} \leftarrow b_{work} \wedge \neg b_{uipcap} \wedge \neg b_{litcap} \wedge b_{slot}$   
 $\beta \leftarrow \text{If}(b_{uipcap} \vee b_{litcap} \vee b_{numcap}, 0, \beta)$  // restart on cap-hit  
 $Seen.clear\_set(Touched, b_{act})$  // linear scan of  $Touched$   
 $c_{\ell} \leftarrow \text{If}(b_{slot}, -learn\_slot, \perp)$  // public learned-clause slot  
 $CL.write(c_{\ell}, OutCl, b_{\ell})$   
 $\text{Backtrack}(\beta, b_{work})$   
 $b_{fail} \leftarrow \text{WL.write\_clause}(c_{\ell}, OutCl, b_{\ell})$   
 $\text{AddImplication}(\alpha, c_{\ell}, b_{\ell} \wedge \neg b_{fail})$   
**return**  $b_{unsat}, b_{fail}$ 

---