ppSAT: Towards Two-Party Private SAT Solving

Ning Luo
Yale University

Samuel Judson
Yale University

Timos Antonopoulos
Yale University

Ruzica Piskac
Yale University

Xiao Wang
Northwestern University

Abstract

We design and implement a privacy-preserving Boolean satisfiability (ppSAT) solver, which allows mutually distrustful parties to evaluate the conjunction of their input formulas while maintaining privacy. We first define a family of security guarantees reconcilable with the (known) exponential complexity of SAT solving, and then construct an oblivious variant of the classic DPLL algorithm which can be integrated with existing secure two-party computation (2PC) techniques. We further observe that most known SAT solving heuristics are unsuitable for 2PC, as they are highly data-dependent in order to minimize the number of exploration steps. Faced with how best to trade off between the number of steps and the cost of obliviously executing each one, we design three efficient oblivious heuristics, one deterministic and two randomized. As a result of this effort we are able to evaluate our ppSAT solver on small but practical instances arising from the haplotype inference problem in bioinformatics. We conclude by looking towards future directions for making ppSAT solving more practical, most especially the integration of conflict-driven clause learning (CDCL).

1 Introduction

Boolean satisfiability (SAT) is a foundational problem in computer science [11, 13, 14, 32]. SAT asks whether there is a variable assignment (or *model*, $\mathcal{M}$) that makes a Boolean propositional formula ϕ evaluate to true. A SAT solver is a tool that takes an instance ϕ as input and checks its satisfiability; solvers can also output a model when one exists. The SAT problem is NP-complete and widely believed to require at least superpolynomial, if not exponential, time [31, 53]. Most state-of-the-art SAT solvers are in fact enhancements of the worst-case exponential branch and backtrack algorithm of Davis-Putnam-Logemann-Loveland (DPLL) [13, 14]. Nonetheless, well-engineered modern solvers such as Kissat, the basis for the winner of the 2021 SAT competition [2], can efficiently resolve large and complex SAT instances containing tens of millions of variables and clauses [2, 6, 7, 20, 26, 48, 54, 55, 56], arising from within program verification [24, 35, 47], networks [5, 39], and numerous other domains [40, 41, 55].

All existing SAT solvers are designed for execution by a single party possessing complete information on ϕ.[1] However, in certain settings SAT instances arise as the conjunction of inputs from two or more distinct parties, *i.e.*, $\phi \equiv \bigwedge_{i\in[k]} \phi_i$ where party P_i formulates ϕ_i independently of ϕ_j for all $j \neq i$. If the P_i are mutually distrustful and their inputs are valuable, privileged, or legally encumbered – and so must be kept private from the other parties – then those solvers are no longer applicable without a trusted intermediary or secure execution environment to run them. In settings without recourse to such trust assumptions, secure computation techniques are instead required to privately resolve SAT instances. Our work combines recent advancements in oblivious algorithm design [60] with classic techniques for SAT solving [13, 14] and secure two-party/multiparty computation (2PC) [61] to develop a solver for *privacy-preserving Boolean satisfiability*, or ppSAT. We also consider the promise and challenge of augmenting this secure computation with differential privacy (DP) [17, 18] to trade off privacy and efficiency, following a strand of recent research [23, 28].

SAT solvers take as input Boolean formulas in conjunctive (also known as clausal) normal form (CNF). We focus on the setting with two parties P_0 and P_1 and a Boolean formula $\phi \equiv \phi_0 \wedge \phi_1 \wedge \phi_{pub}$ such that ϕ_b is the private input of P_b for $b \in \{0, 1\}$ and ϕ_{pub} is an optional public input. This allows us to use the most concretely efficient designs and software for secure computation available. Also, the general architecture and optimizations of our construction should be extendable to support more than two parties given suitable secure computation primitives. We assume that P_0 and P_1 have agreed on the meaning of a set of n variables $v_1, \ldots, v_n$. A two-party ppSAT solver takes private inputs ϕ_0 and ϕ_1 from P_0 and P_1 respectively, where ϕ_b is over the v_i with $|\phi_b| = m_b$ clauses. The solver should correctly output a bit $s = (\exists \mathcal{M}.\ \mathcal{M} \models \phi)$, and optionally output a satisfying $\mathcal{M}$ when possible. Intu-

[1] Prior work does consider parallel/distributed SAT solving, but only where that single party coordinates networked computing resources [46].

itively, we desire a security guarantee that P_b learns nothing more about ϕ_{1-b} than is implied by s, ϕ_b, $|\phi_{1-b}|$, (when input) ϕ_{pub}, and (when output) $\mathcal{M}$.[2] In this paper we design and implement a sound ppSAT solver that meets a slightly relaxed security guarantee, necessary due to the exponential worst-case runtime of our SAT decision procedure. We introduce this weakening formally in §3 and Appendix A.

A Motivating Example. In bioinformatics, inference of haplotypes can uncover genetic information valuable to biological research and medical treatment. Haplotypes are DNA sequences which originate from a single parent, and their information can aid studies on, *e.g.*, genetic risk factors for cardiovascular disease [52] or the effective use of medications [22]. However, current genetic sequencing technology usually only resolves genotypes, which are mixtures of haplotypes from both parents. Given a set of genotypes G drawn from a population, the process of inferring a set of haplotypes H whose elements explain every $g \in G$ (*i.e.*, that are plausibly the biologically-realized haplotypes in that population) is called haplotype inference [22, 25]. One computational approach is haplotype inference by pure parsimony (HIPP) [22, 25]. HIPP finds a minimally-sized H to explain an input G, and is known to be APX-hard [34].

Formally, (sections of) both haplotypes and genotypes may be expressed as strings of length ℓ, with a haplotype $h \in \{0, 1\}^{\ell}$ and a genotype $g \in \{0, 1, 2\}^{\ell}$. Given a pair of haplotypes $hs = (h^0, h^1)$ and a genotype g, the predicate

$$\mathit{explainI}(hs, g) \iff \forall i \in [\ell].\ (h^0_i = h^1_i = g_i) \vee (h^0_i \neq h^1_i \wedge g_i = 2)$$

is true iff g can be explained as a mixture of h^0 and h^1. A set of haplotypes H explains a set of genotypes G if every $g \in G$ can be obtained by pairing two $h^0, h^1 \in H$:

$$\mathit{explain}(H, G) \iff \forall g \in G.\ \exists hs \in H \times H.\ \mathit{explainI}(hs, g).$$

For example, $H = \{010, 110, 001\}$ explains $G = \{210, 022\}$, as $(010, 110)$ explains 210 while $(010, 001)$ explains 022. It is straightforward to see that a minimally-sized H has $2 \leq |H| \leq 2|G|$.

SHIPs [40, 41] solves the HIPP problem for genotype set G by invoking a SAT solver to find an H which makes $\mathit{explain}(H, G)$ true. Specifically, for a conjectured size of the haplotype set $r \in [2, 2|G|]$, the SHIPs algorithm converts $\mathit{explain}(H, G) \wedge |H| = r$ into a CNF formula ϕ where the elements of H are represented by $r \cdot \ell$ variables. The satisfiability of ϕ is then checked by a SAT solver, and if true the resultant model $\mathcal{M}$ encodes H. To find a minimal H, SHIPs starts with $r = 2$ and increments it until ϕ is satisfiable (as a model necessarily exists when $r = 2|G|$).

Genetic information such as genotypes can be expensive to obtain, can carry significant privacy risk, and/or can be encumbered with legal and regulatory protections. Commercial and academic researchers who collect and analyze genotype data often have strong incentives to control access to it [57]. Anonymization and summarization of genetic data is not a panacea, to the extent those notions are technically and legally meaningful at all [49]. Homer *et al.*'s attack [29, 58] shows auxillary information paired with the genotype of a target may be used to identify their participation in a genome-wide association study (GWAS). If two parties respectively hold databases of genotypes G_0 and G_1 and want to run haplotype inference over $G = G_0 \cup G_1$ without exposing their data to the other participant, current technologies for HIPP require trading off the privacy of that data against the economic and social value of the research. Even when legal infrastructure can provide and enforce privacy guarantees to allow otherwise reticent parties to share data, the time and cost of lawyers and negotiations and contracts may very well outpace even the most expensive secure computations.

The application of SHIPs through a ppSAT solver could help mitigate all of this tension. Up to a minor optimization not required for correctness, the CNF formula ϕ encoding $\mathit{explain}(H, G) \wedge |H| = r$ is naturally composed of (i) a public ϕ_{pub} encoding $|H| = r$; and (ii) two independent ϕ_b each derived only from G_b, such that $\mathcal{M} \models \phi$ iff $\mathcal{M} \models \phi_0 \wedge \phi_1 \wedge \phi_{pub}$. As such, each party P_b can construct ϕ_b locally, at which point they may jointly execute the ppSAT solver over ϕ. Solving this formula infers an H that explains G while keeping G_0 and G_1 private, up to their cardinalities. This approach may not completely mitigate privacy concerns, as H is itself (inferred) genetic data which may be, *e.g.* correlated with observable medical conditions and the community of origin of the individuals who provided G. However, as haplotypes are far less diverse within a population, their exposure may carry less risk than that of the underlying genotypes [12, 50].

Naive Approaches. While to the best of our knowledge no specific study of privacy-preserving SAT solving exists in the literature, both basic secure computation primitives and general completeness results can be used to naively instantiate ppSAT solvers. One immediate approach is to use private set disjointedness (PSD) testing [19, 33]. Each P_b could enumerate the set $M_b = \{\mathcal{M}_i \mid \mathcal{M}_i \models \phi_b\}$ of satisfying assignments for their formula, and then the parties could jointly execute a PSD protocol to check whether $|M_0 \cap M_1| > 0$. A model $\mathcal{M}$ could be recovered by replacing PSD with private set intersection (PSI) [51]. However, $|M_b|$ can be worst-case exponential in $|\phi_b|$, is often very large in concrete terms [3, 4], and is leaked by this approach, as is $|M_0 \cap M_1|$ when using PSI. In general this enumeration is #P-hard [7], so straightforward application of PSD/PSI will likely be impractical.

Another naive approach is to raise a preexisting SAT solver to a ppSAT solver through a generic 2PC compiler. However, current such compilers (often bluntly) apply techniques such

[2] As is common in 2PC it is difficult for P_{1-b} to hide the length $|\phi_{1-b}|$. When necessary SAT instances can be padded out with tautological clauses.

as the padding out of loops and linear scan array lookups to create data-oblivious execution paths [27], which will likely be impractical given the amount of state management and data-dependent processing of known SAT decision procedures. Adoption of RAM-based 2PC methods [21] is more promising, but generic use of their compilers is for the moment unreasonably expensive.

1.1 Notation

We denote the two parties as P_0 and P_1, whose inputs are CNF formulas ϕ_0 and ϕ_1 respectively. When applicable we represent a public component of the formula, known to both parties, by ϕ_{pub}. The set of variables in ϕ is $V = \{v_1, \dots, v_n\}$, so $|V| = n$. The number of clauses of an input subformula is $|\phi_b| = m_b$, so that $|\phi| = m = m_0 + m_1 + m_{pub}$. Each clause is composed of the logical disjunction of literals, each of which is either v_i or $\neg v_i$ for $v_i \in V$. We compute satisfiability over $\phi \equiv \phi_0 \wedge \phi_1 \wedge \phi_{pub}$ where the last term appears only as appropriate. A model $\mathcal{M} \in \{0, 1\}^n$ is a function $\mathcal{M} : V \to \{0, 1\}$ mapping variables to truth values. When referring to the satisfiablity value s output by the ppSAT solver, we will often represent $s = 0$ (resp. $s = 1$) by UNSAT (resp. SAT), *i.e.*, UNSAT indicates that there is no satisfying model for ϕ, while SAT indicates the opposite. In practice s will not necessarily be a bit, as we let $s = -1$ indicate that the satisfiability of ϕ is unknown. We notate access to the ith element in x by $x[i]$, and use e_i to represent the unit vector such that $e_i[i] = 1$ and $e_i[j] = 0$ for all $j \neq i$. Finally, the notation $\Pi(a \parallel b; c)$ denotes the execution of a two-party protocol Π with private inputs a and b and public input c.

1.2 Challenges and Contributions

We found constructing a ppSAT solver to require addressing three main challenges.

Data-Oblivious Execution. All 2PC constructions use data-oblivious execution patterns to prevent information leakage. Designing an oblivious version of a SAT decision procedure such as DPLL is difficult as known algorithms and their underlying data structures are highly data-dependent, even for the most basic of operations. For example, such algorithms assume constant-time methods for checking and modifying the inclusion of literals within clauses [44]. Any design must address how to represent clauses so as to allow as efficient oblivious lookup and alteration as possible. DPLL-based SAT solvers also guess and backtrack frequently, implicitly building a search tree dependent upon the input formula. A ppSAT solver must somehow obfuscate the structure of these trees, which requires hiding when guesses and backtracking occur.

Our approach to this challenge is to use a pair of binary matrices (P, N) to encode the formula. In Appendix B we also briefly present a specialized approach for when every clause in ϕ has far fewer than n variables. In general we only ever access individual columns (*i.e.*, clauses) of these matrices in isolation, and so often describe them as vectors of vectors instead of as matrices. When $P_{ij} = P_j[i] = 1$ variable v_i appears in the j-th clause, while $N_{ij} = N_j[i] = 1$ indicates $\neg v_i$ does; the j-th column vectors in P and N together encode the j-th clause.

We also use a pair of binary vectors $(\mathsf{ind}^+, \mathsf{ind}^-)$ to encode literals: v_i is encoded as $(e_i, 0^n)$, and $\neg v_i$ by $(0^n, e_i)$. Negation of a literal is done by just swapping $(\mathsf{ind}^+, \mathsf{ind}^-)$ into $(\mathsf{ind}^-, \mathsf{ind}^+)$. This representation enables oblivious checking of the inclusion of a literal in a clause through a linear-time cascade. We also use the same two-vector encoding for variable assignments. Simplifying clauses is performed by updating P and N according to a comparison with an assignment $a = (\mathsf{ind}^+, \mathsf{ind}^-)$. Finally, we adopt an oblivious stack for guessing and backtracking, in order to hide the search tree.

Heuristics for Guessing. Designing oblivious variants of DPLL subroutines often requires no more than one-to-one translation using techniques such as linear scans and oblivious data structures. However, certain components require more craft and care, most especially the decision heuristics. DPLL searches the space of variable assignments by (i) taking forced choices when possible, (ii) using heuristics to guide unforced choices, and then (iii) unwinding these choices to backtrack when necessary. Much of the unreasonable effectiveness of modern SAT solvers stems from intelligent heuristics [20]. However, these heuristics often rely on data accesses or arithmetic computations that are expensive or impractical in 2PC. For example, a classic heuristic is to just randomly assign a randomly chosen variable that does not yet have a valuation [13]. With constant-time lookups and assignments this is easy, as the set of unassigned literals can efficiently be tracked and randomly sampled from by the SAT solver. However, even this simple heuristic is hard to realize within a ppSAT solver. Nonetheless, we design and implement this and one other randomized guessing heuristic, along with a further deterministic one. Though basic they provide a sound foundation. In §4.3 we discuss potential ways to integrate more powerful heuristics, such as conflict-driven clause learning (CDCL) [48, 54], an essential component of modern solvers which preemptively closes off impossible paths to drastically reduce the effective size of the search tree.

Our deterministic heuristic simply picks the literal with the greatest frequency, which is straightforward to realize with our matrix encoding by counting and comparing. As for randomized heuristics, they require sampling from a distribution dependent on the formula which must be kept secret. We design a private and efficient method for sampling $\ell^* \leftarrow_{\mathcal{D}} \{\ell_1, \cdots, \ell_k\}$ for any distribution $\mathcal{D}$ that can be expressed as a list of integers $\{w_1, \dots, w_k\}$ such that $\Pr[\ell^* = \ell_i] = \frac{w_i}{\sum_j w_j}$, which is closely related to prior work in the literature [9]. Using this technique we can instantiate a randomized heuristic that selects an unassigned literal with, *e.g.*, probability uniform or proportional to its frequency.

Leakage vs. Efficiency. Recall that our natural security requirement is that P_b learns nothing more about ϕ_{1-b} than is implied by s, ϕ_b, $|\phi_{1-b}|$, (when input) ϕ_{pub}, and (when output) $\mathcal{M}$. However, if formalized such a definition would be slightly too strong to be practical. To prevent information leakage from the runtime of the ppSAT solver, meeting this guarantee would require it to *always* run in time $T(\lambda, n, m)$ for security parameter λ and deterministic function T. Since all known SAT decision procedures have worst-case runtime bounds $O(g(n))$ for $g(n) > (1+k)^n$ for constant $k > 0$ [7], completeness would require T be exponential in its inputs. This behavior would both (i) be impractical for all but very small n; and (ii) likely conflict with assumptions underlying the security arguments for the 2PC primitives we need [38, 60].

As such we will sacrifice completeness. Instead, we assume the parties agree on some polynomial $\tilde{T}(x, y, z)$ and set $\tilde{T}(\lambda, n, m) = \tau_{\lambda,n,m}$ as the upper-bound on the runtime of the solver, based, *e.g.*, on an economic or social analysis of the value of solving the instance for that concrete cost. This polynomial upper-bound mitigates our concern about the security of our underlying cryptographic primitives. The parties can then agree to either (i) run for $\tau_{\lambda,n,m}$ steps always, aborting if that is insufficient to resolve the instance, or (ii) to terminate upon resolution at time $\tau \leq \tau_{\lambda,n,m}$, with an accordant risk of information leakage.

This tradeoff between the most efficient possible execution of the ppSAT solver and the greatest possible privacy must be jointly agreed upon by the P_b. We will generally focus on the case of (ii), and discuss how techniques from differential privacy (DP) [17, 18] may be used to add calibrated noise to reduce the informational content of this leakage without overly extending the runtime of the solver. We formalize security definitions for (i) and (ii) without differential privacy in §3 and Appendix A, and the latter with DP in Appendix D. All three of these guarantees are weaker than the natural security definition, since the adversary can learn additional information in the form of one of (i) τ, (ii) τ with added noise, or (iii) that $\tau_{\lambda,n,m}$ steps were insufficient to resolve the instance.

2 Preliminaries

2.1 Overview of DPLL

To illustrate how the DPLL procedure works we walk through its execution on an example. Consider the following Boolean formula with four variables $\phi^{(0)} \equiv (v_1 \vee v_2) \wedge (v_2 \vee \neg v_3 \vee v_4) \wedge (\neg v_1 \vee \neg v_2) \wedge (\neg v_1 \vee \neg v_3 \vee \neg v_4) \wedge (v_1)$. The procedure iteratively builds a model for the formula by repeating a sequence of steps. As the input formula is in CNF its model must be a model for (*i.e.*, satisfy) each of its clauses.

We start with the empty model, $\mathcal{M} = \varnothing$. The first step, UNITSEARCH, searches for a unit clause: a clause consisting of a single literal. Assigning a truth value which makes that literal true is the only way to find a model for the formula. In our example the only unit clause is (v_1), the last clause. We add its satisfying assignment to the model: $\mathcal{M} = \{v_1 = 1\}$. This model is not only a model for that last clause, but for the first clause as well; it is a model for every clause where v_1 appears as a positive literal. This observation is the basis for the PROPAGATION step, which is run every time a new element is added to the model. The PROPAGATION step removes that element from the formula: all clauses where v_1 appears positively are removed, while in the remaining clauses we can safely remove the negated v_1 variable as $\neg v_1$ evaluates to false under $\mathcal{M}$, and false is a neutral element in disjunctions. In our particular example it means that we are left with the formula $\phi^{(1)} \equiv (v_2 \vee \neg v_3 \vee v_4) \wedge (\neg v_2) \wedge (\neg v_3 \vee \neg v_4)$.

The procedure now again executes the UNITSEARCH step, finding the unit clause $(\neg v_2)$. In general, after every UNITSEARCH step which finds a unit clause ℓ, the procedure executes the CHECK step, which checks that $\neg\ell$ is not also a unit clause. The CHECK step does not find any such conflict here, so we add $\neg v_2$ to the model: $\mathcal{M} = \{v_1 = 1, v_2 = 0\}$.

After the PROPAGATION step we are left with two clauses: $\phi^{(2)} \equiv (\neg v_3 \vee v_4) \wedge (\neg v_3 \vee \neg v_4)$, and running UNITSEARCH does not find a unit clause. When this occurs we pick a variable, guess its value, and then add that to the model. This is a DECISION step. For example, we can add $v_3 = 1$ to the model: $\mathcal{M} = \{v_1 = 1, v_2 = 0, v_3^d = 1\}$. Note that v_3 is annotated with d, indicating that it is a decision variable. Its value was guessed and not inferred. We then run the PROPAGATION step, which results in a new formula $\phi^{(3)} \equiv (v_4) \wedge (\neg v_4)$.

We again run the UNITSEARCH step on $\phi^{(3)}$: now (v_4) is a unit clause. However, running the CHECK step will find that $(\neg v_4)$ is also a unit clause. Therefore we need to backtrack. Backtracking is possible only when there is a decision literal in the model. The BACKTRACK step retreats to the point in the procedure just before the last decision variable was added. Instead, we add its negation to the model and remove the d annotation. It is now inferred that the negated value has to be in the model, otherwise we would derive a contradiction.

Running the BACKTRACK step results in the model $\mathcal{M} = \{v_1 = 1, v_2 = 0, v_3 = 0\}$. We apply PROPAGATION on $\phi^{(2)}$ and the resulting formula is empty, *i.e.*, $\mathcal{M}$ is a model for all clauses in the original formula, which means that $\phi^{(0)}$ is satisfiable and $\mathcal{M}$ is its model. Note that v_4 can have any value. When the CHECK step finds a contradiction and no prior guesses can be undone, DPLL terminates and reports that the original input formula is unsatisfiable.

2.2 Cryptographic Preliminaries

Basic Primitives. We use standard techniques for 2PC, wherein P_0 and P_1 employ binary garbled circuits (GC) built from oblivious transfer (OT) and encryption primitives to jointly compute the sequence of functionalities which make up our ppSAT solver. Our design guarantees the order of these functionalities is fixed (up to the length of the execution), and that the access patterns over all intermediary values are data-oblivious, *i.e.*, either fixed or randomized independently of the private protocol inputs (up to their length). Those intermediary values are then stored at rest distributed between the P_b with information-theoretic security. At the end of the protocol, the final outputs are revealed to both parties.

Oblivious Stack. An oblivious stack data structure allows for conditional operations, which take a secret Boolean value that dictates whether the operation is actually performed or simulated through a dummy execution [60]. We will rely on the following operations, where $\bot$ and $\bot'$ are arbitrary but distinguishable special symbols:

- $\mathsf{ObStack} \leftarrow \mathsf{stack}()$: initialize an oblivious stack;
- $(\cdot) \leftarrow \mathsf{ObStack.CondPush}(b, x)$: (conditionally) push element x to the oblivious stack if $b = 1$, else skip.
- $(x) \leftarrow \mathsf{ObStack.CondPop}(b)$: (conditionally) pop and return the top element x if $b = 1$, else return $\bot$. If $b = 1$ and the stack is empty, fail and output $\bot'$.

3 Overview

As noted, a complete ppSAT solver can run for exponential time, which violates standard 2PC security definitions. In this section we formulate a definition for which meaningful security is practically achievable. We then give a high-level overview of our solver, before presenting it in detail in §4.

3.1 Formalizing ppSAT Security

A ppSAT solver is a two-party secure computation (2PC) protocol executed by P_0 and P_1. We operate in the *semi-honest* model, *i.e.*, we consider an adversarial P_b which attempts to learn private information about ϕ_{1-b}, but does not otherwise deviate from the protocol. We formalize security under the simulation paradigm [10]. The *view* of party P_b is an object containing all information known to it at the conclusion of a protocol Π: its private and the public inputs, every random coin flip it samples, every message it receives from P_{1-b}, every intermediary value it computes, and the output. We consider the protocol secure if there exists a *simulator* Sim_{1-b} such that no efficient algorithm $\mathcal{A}$ can distinguish between the view of P_b when interacting with Sim_{1-b} in an *ideal world* vs. with P_{1-b} in the *real world*. The simulator is given only (i) the private inputs of P_b, (ii) the public inputs, and (iii) the output of the protocol. Since the view of P_b in the ideal world cannot directly contain any information about ϕ_{1-b} by definition, this indistinguishability implies an adversarial P_b cannot learn more about it than what is implied by the output in the real world either.

However, standard computational security definitions for simulation are not blindly applicable to ppSAT solving as they require all parties run in probabilistic-polynomial time (PPT), while a complete solver may require exponential time with non-negligible probability. As such, to retain these definitions we choose to yield completeness for our solver and force polynomial runtimes, at the further cost of information leakage. In Appendix A we rigorously formalize four different variants of a simulation-based security definition for ppSAT solving. Each requires some leakage, but how much depends on (i) whether the running time is leaked to allow early termination; and (ii) whether a model is output. We will primarily focus on a two-party-exact-time-revealing solver ($2p$-etr-solver), which reveals only (i) and so is the most efficient and concise formulation. We also discuss a further modification of the definition permitting the use of differential privacy to hide some of the information leakage in Appendix D.

3.2 Oblivious DPLL

Our ppSAT solver consists of a sequence of *giant steps* implementing an oblivious variant of the DPLL procedure. For concision we walk through solving without $\mathcal{M}$, and briefly discuss its addition at the end. At initialization the solver sets $\phi^{(0)} \leftarrow \phi$ and $a^{(0)} \leftarrow \bot$, where the latter is a "dummy value" encoded as a pair $(0^n, 0^n)$. As shown in Figure 1, the t-th giant step takes as input a formula $\phi^{(t-1)}$ and an assignment $a^{(t-1)}$. Each giant step either returns SAT/UNSAT or outputs an updated formula $\phi^{(t)}$ and a single assignment $a^{(t)}$ for the next giant step to consume.

A giant step sequentially executes oblivious variants of the five core small step algorithms of the DPLL procedure: UNITSEARCH, DECISION, CHECK, BACKTRACK, and PROPAGATION. First, UNITSEARCH and then DECISION output an assignment, $a^{(t)}_{\mathsf{unit}}$ and $a^{(t)}_{\mathsf{dec}}$ respectively. The UNITSEARCH routine scans $\phi^{(t-1)}$ and (when one exists) finds a unit clause. If such a clause is found then $a^{(t)}_{\mathsf{unit}}$ encodes its single literal as an assignment, otherwise it encodes the dummy value $\bot$. The DECISION routine invokes a chosen heuristic (usually, but not necessarily, fixed for the entire execution) and obtains $a^{(t)}_{\mathsf{dec}}$ as a guess. If $a^{(t-1)} = a^{(t)}_{\mathsf{unit}} = \bot$ then $(\phi^{(t-1)}, a^{(t)}_{\mathsf{dec}})$ is pushed onto the oblivious stack. Otherwise a dummy push operation is performed. The multiplexer Mux_0 then selects a non-dummy assignment according to the priority $a^{(t-1)} > a^{(t)}_{\mathsf{unit}} > a^{(t)}_{\mathsf{dec}}$. Note that $a^{(t)}_{\mathsf{dec}} \neq \bot$ always. The selected assignment, denoted $a^{(t)}_{\mathsf{sel}}$, and $\phi^{(t-1)}$ are then taken by the CHECK routine as input.

This routine is the only possible point when the procedure can terminate and return SAT/UNSAT. For the moment we assume the procedure terminates immediately when possible, and discuss alternative behaviors later. A CNF formula conflicts with an assignment if it leads to an unsatisfiable clause. For input $\phi^{(t-1)}$ and assignment $a^{(t)}_{\mathsf{sel}}$ there are four possible cases for CHECK, PROPAGATION, and BACKTRACK:

1. The CHECK subroutine finds that $\phi^{(t-1)}$ is satisfied and returns SAT.
2. The CHECK subroutine finds that $\phi^{(t-1)}$ conflicts with $a^{(t)}_{\mathsf{sel}}$ and the stack is empty, and so returns UNSAT.
3. No conflict occurs, so CHECK passes $\phi^{(t-1)}$ and $a^{(t)}_{\mathsf{sel}}$ to the PROPAGATION and BACKTRACK routines. The PROP-

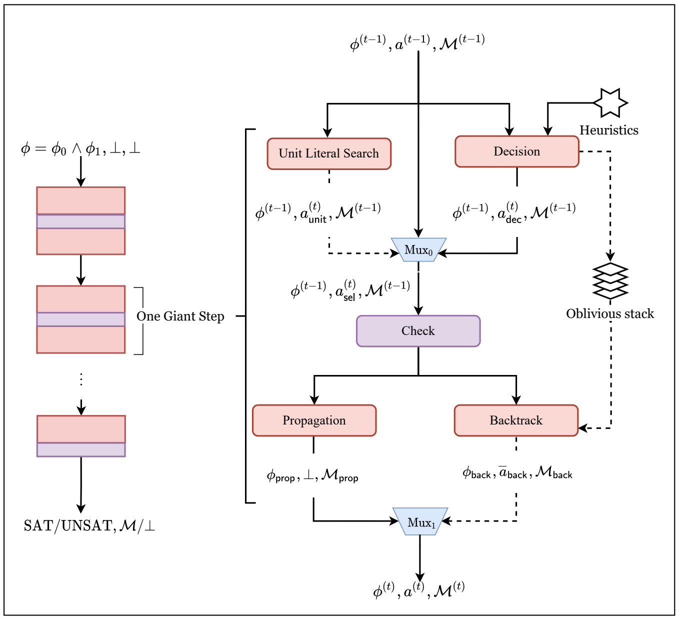
$\phi^{(t-1)}, a^{(t-1)}, \mathcal{M}^{(t-1)}$
$\phi = \phi_0 \wedge \phi_1, \bot, \bot$
One Giant Step
SAT/UNSAT, $\mathcal{M}/\bot$
Unit Literal Search
Decision
Heuristics
$\phi^{(t-1)}, a^{(t)}_{\mathsf{unit}}, \mathcal{M}^{(t-1)}$
$\phi^{(t-1)}, a^{(t)}_{\mathsf{dec}}, \mathcal{M}^{(t-1)}$
Mux_0
$\phi^{(t-1)}, a^{(t)}_{\mathsf{sel}}, \mathcal{M}^{(t-1)}$
Oblivious stack
Check
Propagation
Backtrack
$\phi_{\mathsf{prop}}, \bot, \mathcal{M}_{\mathsf{prop}}$
$\phi_{\mathsf{back}}, \bar{a}_{\mathsf{back}}, \mathcal{M}_{\mathsf{back}}$
Mux_1
$\phi^{(t)}, a^{(t)}, \mathcal{M}^{(t)}$

Figure 1: The structure of a giant step and its role in our ppSAT solver, demonstrating the high-level design given in §3. Dashed arrows indicate potential dummy return values.

AGATION routine simplifies $\phi^{(t-1)}$ to ϕ_{prop} by eliminating both clauses which have been satisfied by $a^{(t)}_{\mathsf{sel}}$ and literals $\neg a^{(t)}_{\mathsf{sel}}$. The Mux_1 multiplexer will then set $\phi^{(t)} \leftarrow \phi_{\mathsf{prop}}$ and $a^{(t)} \leftarrow \bot$ as the output of the giant step. The BACKTRACK routine will execute a dummy pop operation.

4. A conflict occurs and the stack is not empty. The BACKTRACK routine pops a formula $\phi_{\mathsf{back}} = \phi^{(t')}$ for some $t' < t-1$ and its associated a_{back} from the stack. It then sets $\bar{a}_{\mathsf{back}}$ by swapping ind^- and ind^+ from a_{back}. The Mux_1 multiplexer will then select $\phi^{(t)} \leftarrow \phi_{\mathsf{back}}$ and $a^{(t)} \leftarrow \bar{a}_{\mathsf{back}}$ as the output of the giant step. The output of the PROPAGATION routine will be ignored.

When the model $\mathcal{M}$ is desired as output this design is easily modified to continually update its state during the PROPAGATION routine, as well as save that state within and retrieve it from the stack during backtracking.

4 A ppSAT Solver

In this section we formalize the algorithm sketched in §3 as the basis for our ppSAT solver protocol. We begin by defining a pair of abstract data structures for a formula ϕ and its constituent clauses, and then describe an instantiation of these objects and their operations using bit-vectors. These operations are all data-oblivious and include most of the functionalities that will be computed using garbled circuits when the overall design is raised into a secure computation protocol. Finally, we describe how our ppSAT solver is structured as a data-oblivious sequence of these operations. For concision and clarity we sometimes describe the solver with data-dependent branching, but all conditions are simple checks of Boolean variables which can be merged into the operations they guard.

4.1 Data Structures for CNF Formulas

Let β be an integer and $L = \{\ell_1, \ldots, \ell_{2n}\}$ be a set of literals. A clause is a subset $C \subseteq L$ for which $|C| \leq \beta$ and no two distinct literals reference the same underlying variable – implying that $\beta \leq n$ is the maximum clause length in ϕ. This parameter allows us to design different instantiations for when it is public knowledge that $\beta \approx n$, as opposed to when, *e.g.*, $\beta \ll n$. We require three operations over clauses:

- $C.\mathsf{unit}()$: returns a bit indicating whether $|C| = 1$.
- $C.\mathsf{contain}(\ell)$: returns a bit indicating whether $\ell \in C$.
- $C.\mathsf{remove}(a)$: updates C to $C \setminus \{a\}$ in place.

A formula ϕ is composed of a set of clauses $\{C_1, \cdots, C_m\}$, for which we also need two operations:

- $\phi.\mathsf{empty}()$: returns a bit indicating whether $\phi = \varnothing$.
- $\phi.\mathsf{remove}(C_j)$: updates ϕ to $\phi \setminus \{C_j\}$ in place.

Instantiating these abstract data structures for a given β requires both state encodings and supporting these methods.

Instantiating the ADS for $\beta \approx n$. Recall that e_i is the unit vector where $e_i[i] = 1$ and $e_i[j] = 0$ for all $j \neq i$. A literal ℓ is represented by the pairing of a unit vector and zero vector $(\mathsf{ind}^+, \mathsf{ind}^-)$. A positive variable v_i is encoded as $(e_i, 0^n)$, while its negation $\neg v_i$ is encoded as $(0^n, e_i)$. A dummy assignment $\bot$ is represented by $(0^n, 0^n)$. The representation of variables cooperates with the encoding of clauses (see next paragraph) so that operations over them can be implemented using only linear scans.

A clause $C_j \in \phi$ is represented by an integer n_L and the pair of vectors (P_j, N_j) such that

$$(P_j[i], N_j[i]) = \begin{cases} (1,0) & \text{if } v_i \text{ appears in } C_j \\ (0,1) & \text{if } \neg v_i \text{ appears in } C_j \\ (0,0) & \text{o.w.} \end{cases}$$

The integer n_L is used to track the number of literals in the clause. The implementation of the three clausal operations are given in Algorithm 1. Determining whether C_j is a unit clause can be implemented by checking whether $n_L = 1$. To implement contain we use that the structure of the clause and literal vectors provides that $\ell = v \in C_j$ iff $\bigvee_{i=1}^{n} (P_j[i] \wedge \mathsf{ind}^+[i]) = 1$ and similarly for $\ell = \neg v$ using N_j and ind^-. To remove a literal v_i (resp. $\neg v_i$) from C_j due to an assignment requires setting $P_j[i] = 0$ (resp. $N_j[i] = 0$) and deducting from n_L. Given the indicating assignment a, the former is the same as updating $P_j[i] \leftarrow P_j[i] \wedge (P_j[i] \oplus \mathsf{ind}^+[i])$ for each $i \in [n]$, and similarly over N_j and ind^-.

Algorithm 1: Clausal Algorithms when $\beta \approx n$

```
Function $C_j$.unit():
    return $(n_L = 1)$
Function $C_j$.contain($\ell = (ind^+, ind^-)$):
    $b \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
        $b \leftarrow b \lor (P_j[i] \land \mathsf{ind}^+[i]) \lor (N_j[i] \land \mathsf{ind}^-[i])$
    return b
Function $C_j$.remove($a = (ind^+, ind^-)$):
    if $C_j$.contain $(a)$ then
        $n_L \leftarrow n_L - 1$
    for $i \in [n]$ do
        $P_j[i] \leftarrow P_j[i] \land (P_j[i] \oplus \mathsf{ind}^+[i])$
        $N_j[i] \leftarrow N_j[i] \land (N_j[i] \oplus \mathsf{ind}^-[i])$
```

A formula ϕ is encoded as matrices (P,N) where the jth column of P is P_j and of N is N_j, as well as a vector $\mathsf{isAlive} \in \{0,1\}^m$ whose jth entry indicates whether C_j has been removed from the formula. The $\phi.\mathsf{remove}(C)$ functionality can be implemented by setting $\mathsf{isAlive}[j] = 0$ during a linear scan, while $\phi.\mathsf{empty}()$ by checking whether $\mathsf{isAlive} = 0^n$. We omit their formal descriptions due to their simplicity.

Alternate Approaches. We primarily focus on the above approach, as it is a generic solution applicable to every possible ppSAT instance. We also consider an alternate instantiation for when the maximum number of literals in a clause is much smaller than n (*i.e.*, $\beta \ll n$) in Appendix B.

Operations on clauses can in theory be instantiated via RAM-model secure computation [21], which requires running an ORAM client algorithm in MPC. This could potentially reduce the asymptotic cost of the ADS operations to $O(\log^2 n)$ from $O(n)$.[3] We can empirically compare our bit-vector based solution with the most practically efficient ORAM secure computation [16]. To read or write a bit from a n-bit vector the linear scan circuit contains exactly $2n-1$ AND gates. As a result, the crossover point (conservatively) occurs when $n \approx 2^{17}$, where our circuit takes about 0.13s and the ORAM-based solution takes about 0.2s.

4.2 Data-Oblivious ppSAT Solving

Algorithm 2 formally presents the algorithmic structure of our ppSAT solver. We only provide a full description of our *2p-etr*-solver for brevity, which can be extended to support $\mathcal{M}$ as described in §3, and raised into a secure 2PC protocol using the standard techniques referenced in §2. Finally, we abuse notation by considering τ and $\tau_{\lambda,n,m}$ to track and upper bound respectively the number of giant steps. Their true values are some constant factor (representing the number of computational steps within a giant step) of how we use them algorithmically.

Every giant step (Lines 13-24 and 3-12 across two loop iterations) starts with a formula ϕ and an assignment a, and either passes a new formula and assignment to the next giant step or terminates. The flag b_{conflict} indicates a conflict; when one (and therefore backtracking) occurred in the prior giant step then $b_{\mathsf{conflict}} = 1$.

The solver first executes UNITSEARCH, sets b_{unit} to indicate its success, and if successful returns the unit literal as an assignment a_{unit}. Then the solver invokes the heuristic in DECISION and receives a branching assignment a_{dec}. If $b_{\mathsf{conflict}} = 1$ the negation of a_{back} and ϕ_{dec} from the previous giant step are taken as the input of CHECK (Lines 21-23 and 4). Otherwise, either the output of UNITSEARCH or the output of DECISION will be used depending on b_{unit} (Lines 16-19 and 4).

The CHECK routine resolves the application of assignment a to ϕ. There are three possibilities, each corresponding to a value of σ:

1. $\sigma = 0$: ϕ is satisfied after applying a. Then CHECK terminates the procedure and outputs SAT (Line 6);

2. $\sigma = 1$: ϕ contains a unit clause with the negation of a (Line 7). The solver then pops the top element (if any) off the stack (Line 8). If the stack is empty the solver will terminate and output UNSAT (Lines 9-10). Otherwise, b_{conflict} is set to 1 and the solver backtracks, ultimately recovering the assignment a_{back} and formula ϕ_{back} for the next giant step. The result of PROPAGATION will be ignored; or

3. $\sigma = 2$: the formula is neither SAT nor in conflict after applying a. The PROPAGATION routine simplifies ϕ to ϕ_{prop} using a and passes the simplified formula to the next giant step (Line 12).

The behavior of BACKTRACK is directly integrated into Algorithm 2, while DECISION is the focus of §4.3. Next, we describe the remaining UNITSEARCH, CHECK, and PROPAGATION routes in detail – due to space constraints we give their formal definitions in Appendix C.

UNITSEARCH (Algorithm 8): The UNITSEARCH routine finds a unit clause in the current formula ϕ when one exists, and outputs a bit b and an assignment a. If no unit clause exists in ϕ then $b = 0$, else $b = 1$ and a corresponds to its single literal. To find the encoding of that literal to set a, we need to locate the clause C_j such that $C_j.\mathsf{unit}() = 1$. We achieve this through a linear scan of all the clauses, setting $b \leftarrow 1$ and $a \leftarrow C_j$ once we find a suitable output (Lines 2-5).[4]

[3] Note that although the best ORAM [1] can incur only a $O(\log n)$ overhead, that requires an underlying data block of at least $\log n$ bits. In our case, each data block is a single bit and thus the best available requires $O(\log^2 n)$. We are not aware of any ORAM designed specifically for bit accesses, which may be a promising line of future work to increase its efficacy in this and other secure computations over bit-vectors.

[4] We slightly abuse notation here for readability, as not all instantiations

CHECK (Algorithm 9): The CHECK routine determines whether formula ϕ is SAT, and if not whether the assignment a causes a conflict or whether the resultant ϕ remains viable but unproven. It returns $\sigma \in \{0, 1, 2\}$. The three cases are (i) $\sigma = 0$, indicating that ϕ is satisfied; (ii) $\sigma = 1$, indicating that ϕ conflicts with a; and (iii) $\sigma = 2$, otherwise. The routine uses Boolean variables b_0 and b_1 to track if ϕ is SAT or conflicts with a respectively. The ϕ.empty operation resolves b_0 (Line 1). A clause conflicts with a if it only contains $\neg a$. The routine scans over all clauses, and sets $b_1 \leftarrow 1$ if any is unit and conflicts with the assignment (Lines 3-5). If neither b_0 nor b_1 is set to 1, the routine returns 2 to indicate that ϕ is still viable under a.

PROPAGATION (Algorithm 10): The PROPAGATION routine simplifies ϕ by eliminating clauses containing a literal ℓ with identical indicators to the assignment a. Additionally, $\neg\ell$ is removed from any clause containing it. So, during propagation there are three types of clauses $C \in \phi$, those for which: (i) $\neg a = \ell \in C$, in which case C.remove$(\neg a)$ is executed (Line 7); (ii) $a = \ell \in C$, so C is satisfied and ϕ.remove(C) is therefore invoked (Line 5); or (iii) clause C contains neither $\ell = a$ nor $\ell = \neg a$, and so is left unchanged.

4.3 ppSAT Decision Heuristics

The final component of our solver is the DECISION routine, which is not a single functionality but rather a family of possible procedures for guessing a variable assignment. Designing such techniques is historically a very rich area of SAT research [7, 20, 45], though many of these constructions are not naturally implementable through oblivious computation and 2PC primitives. For this initial work we focus on three heuristics: (i) the deterministic dynamic largest independent sum (DLIS) heuristic, where we choose the most common literal as the assignment, (ii) the randomized (RAND) heuristic where we make a uniform choice over both variable and assignment, and (iii) a weighted randomized heuristic (Weighted-RAND), where the choice of literal is weighted by frequency. These are all relatively simple but still useful, and implementing their many variants along with more complex heuristics is a promising avenue for future work which we discuss at the end of the section.

Note that for brevity we do not provide the full DECISION routine. Each heuristic either returns the assignment a itself or a tuple $d = (i, b)$ encoding that the DECISION routine should construct an assignment by setting $v_i = b$ in the form needed for the given ADS instantiation.

DLIS. The DLIS heuristic selects the most commonly appearing literal and returns the assignment that makes it true. Our formulation of it as Algorithm 3 undertakes a linear scan over every $C_j \in \phi$ for every $v_i \in V$. The heuristic calculates the frequency of v_i and $\neg v_i$ as $\sum_{j=1}^{m} C_j$.contain(v_i) and $\sum_{j=1}^{m} C_j$.contain$(\neg v_i)$ respectively. It then determines whether either of v_i or $\neg v_i$ is the most frequent literal seen so far, and if so sets d as necessary to encode it. After iterating over all the variables d encodes the most frequent literal and is returned.

Algorithm 2: *2p-etr* ppSAT Solver

```
Input: $\phi$, $n$, $m$, $\tau_{\lambda,n,m}$
Output: SAT/UNSAT
ObStack $\leftarrow$ stack(); $\tau \leftarrow 0$; $a \leftarrow \bot$; $b_{\text{conflict}} \leftarrow 0$;
while $\tau \leq \tau_{\lambda,n,m}$ do
    if $\tau \neq 0$ then
        $\sigma \leftarrow$ Check$(\phi, a)$;
        if $\sigma = 0$ then
            return SAT
        $b_{\text{conflict}} \leftarrow (\sigma = 1)$;
        $e \leftarrow$ ObStack.pop$(b_{\text{conflict}})$;
        if $e \leftarrow \bot'$ then
            return UNSAT
        $a_{\text{back}}, \phi_{\text{back}} \leftarrow e$;
        $\phi_{\text{prop}} \leftarrow$ Propagation$(\phi, a)$;
    $(b_{\text{unit}}, a_{\text{unit}}) \leftarrow$ UnitSearch$(\phi_{\text{prop}})$ ;
    $a_{\text{dec}} \leftarrow$ Decision$(\phi_{\text{prop}})$;
    ObStack.CondPush$(\neg b_{\text{unit}} \wedge \neg b_{\text{conflict}}, (a_{\text{dec}}, \phi_{\text{prop}}))$;
    if $b_{unit} = 0$ then
        $a \leftarrow a_{\text{dec}}$;
    else
        $a \leftarrow a_{\text{unit}}$;
    $\phi \leftarrow \phi_{\text{prop}}$;
    if $b_{conflict} = 1$ then
        $a \leftarrow \neg a_{\text{back}}$;
        $\phi \leftarrow \phi_{\text{back}}$;
    $\tau = \tau + 1$;
```

Algorithm 3: DLIS Heuristic

```
Input: $L = \{\ell_1, \ldots, \ell_{2n}\}$, $C = \{C_1, \ldots, C_m\}$
Output: $d \in [1..n] \times \{0, 1\}$
$max \leftarrow 0$; $d \leftarrow (\bot, \bot)$;
for $i \leftarrow 1$ to $n$ do
    $s_P \leftarrow 0$, $s_N \leftarrow 0$;
    for $j \leftarrow 1$ to $m$ do
        $s_P = s_P + C_j$.contain$(\ell_i)$;
        $s_N = s_N + C_j$.contain$(\neg\ell_i)$;
    if $s_N \geq max$ then
        $d \leftarrow (i, 0)$; $max \leftarrow s_N$;
    if $s_P \geq max$ then
        $d \leftarrow (i, 1)$; $max \leftarrow s_P$;
return $d$;
```

RAND. Let the binary vector $\hat{U} \in \{0, 1\}^n$ indicate whether the i-th variable in V has been assigned. The RAND heuristic guesses a variable assignment by uniformly selecting a random unassigned variable and setting it to a bit also chosen

may have unit clause representations which can be used immediately as an assignment. The procedure can be generalized in practice by extending C_j.unit() to return the literal when it is true.

Algorithm 4: Random Value Sampler internal (RVSi)

```
Input: $Q \in \mathbb{N}$, $r'_0 \in \{0,1\}^l$, $r'_1 \in \{0,1\}^l$
Output: $r \in \{0,1\}^l$
$b \leftarrow 0$; $Q' \leftarrow 0^l$;
for $i \in \{l-1, \ldots, 0\}$ do
    if $Q[i] \neq 0 \vee b = 1$ then
        $Q'[i] \leftarrow 1$; $b \leftarrow 1$;
$r' \leftarrow r'_0 \oplus r'_1$;
$r \leftarrow \sum_{i \in [l]} r'[i] \cdot (Q'[i] \cdot 2^i)$;
return $r$;
```

uniformly at random. Since $\hat{U}$ is derived from ϕ as well as prior assignments, the computation in this procedure must be data-oblivious and amenable to efficient realization by secure computation primitives. At the core of our design is Algorithm 4, which with probability $p_1 \geq 1/2$ obliviously selects a secret $r \in [Q]$ for private $Q \in \mathbb{N}$.

We assume that Q is encoded as a binary string of length $l \in \mathbb{N}$. This is the natural encoding for binary garbled circuits, but may not be for other 2PC primitives. We let $h \in \mathbb{N}$ be one greater than the index (from zero) of the most significant non-zero bit in Q, *e.g.*, $h = 4$ for $l = 6$ and $Q = 9 = 001001$. The construction builds a multiplexer which maps an integer $x' \in [2^l]$ to $x \in [2^h]$ by keeping the lower h bits unchanged while setting the upper $l - h$ bits to zero. It then applies this multiplexer to a random binary string $r' \in \{0,1\}^l$, generating $r \in \{0,1\}^l$ to be interpreted as an integer upon return. To sample r' within 2PC we define it as $r' = r'_0 \oplus r'_1$, where r'_b is privately sampled by P_b.

Notice that $r < Q$ with some probability $p_1 \geq 1/2$, as it is guaranteed when $r'[l-h] = 0$. The parties can repeat Algorithm 4 for sufficient $\kappa \in \mathbb{N}$ so that the probability every returned r lies outside $[Q]$, or $p_2 \leq 2^{-\kappa}$, is suitably negligible. A reasonably small constant such as $\kappa = 32$ suffices. A wrapper function RVS(), which we otherwise omit, can make these repeated invocations of RVSi() and then undertake a linear scan over the outputs to finally return, *e.g.*, the last which lies within the desired range.

Our construction to uniformly select an unassigned variable is Algorithm 5. It linearly scans $\hat{U}$ and counts the number of unassigned variables, before invoking Algorithm 4 to get a random index k. It then selects the k-th unassigned variable through another scan of $\hat{U}$, and assigns to it a random $b = b_0 \oplus b_1$, where b_b is sampled and provided by P_b.

Weighted-RAND. Let $L = \{\ell_1, \ldots, \ell_{2n}\}$ be a set of literals and $\mathcal{D}$ a distribution over L expressed as set of positive integers $W = \{w_1, \ldots, w_{2n}\}$ such that for all $i \in [1..2n]$

$$\Pr(\ell_i) = \frac{w_i}{S_W}, \text{ for } S_W = \sum_{j=1}^{2n} w_j.$$

For our decision heuristic w_i will be the frequency count of the i-th literal. We design an oblivious algorithm that randomly samples an element of L according to W in Algorithm 6. First we compute $c = w_1 + \cdots + w_{2n}$. Then, using Algorithm 4 an integer k is sampled from $[c]$. Let $F(\ell_i) = \sum_{j=1}^{i} w_j$. The algorithm finds ℓ_i such that $F(\ell_{i-1}) < k \leq F(\ell_i)$ through a linear scan, which is then returned as the assignment. The intuition behind the correctness of the algorithm is that for any random variable Y, a sample value can be generated by drawing a random $r \in [0,1]$ and then finding its preimage on the cdf of Y.

Algorithm 5: Uniform Random Selection

```
Input: $\hat{U} \in \{0,1\}^n$, $r'_0, r'_1 \in \{0,1\}^l$, $b_0, b_1 \in \{0,1\}$
Output: $d \in [1..n] \times \{0,1\}$
$c \leftarrow 0$; $d \leftarrow (\bot, \bot)$;
for $i \leftarrow 1$ to $n$ do
    $c \leftarrow c + \hat{U}[i]$;
$k \leftarrow$RVS$(c, r'_0, r'_1)$;
for $i \leftarrow 1$ to $n$ do
    $k \leftarrow k - \hat{U}[i]$;
    if $k = 0$ then
        $d \leftarrow (i, b_0 \oplus b_1)$;
return $d$;
```

Algorithm 6: Weighted Random Selection

```
Input: $W \in \mathbb{Z}^{2n}_{\geq 0}$, $L = \{\ell_1, \ldots, \ell_{2n}\}$, $r'_0, r'_1 \in \{0,1\}^l$
Output: $a = (\mathsf{ind}^+, \mathsf{ind}^-)$
$c \leftarrow 0$;
for $i \leftarrow 1$ to $2n$ do
    $c = c + w_i$;
$k \leftarrow$RVS$(c, r'_0, r'_1)$;
for $i \leftarrow 1$ to $2n$ do
    if $0 < k < w_i$ then
        $a \leftarrow \ell_i$;
    $k \leftarrow k - w_i$;
return $a$;
```

CDCL. Perhaps the most important development in SAT solving since DPLL itself is conflict-driven clause learning (CDCL) [20, 48, 54]. The essential idea of CDCL is that whenever a conflict is found it is possible to resolve a self-contained subset of the assignments which triggered the conflict. For example, the CDCL learning procedure might learn that $x_1 = 1, x_{12} = 0, x_{27} = 1$ is impossible for any model. So, by creating a new clause made out of their negation, *i.e.*, $(\neg x_1 \vee x_{12} \vee \neg x_{27})$ and adding it to ϕ, any branches of the search tree which would try that impossible combination are cut off immediately. This dramatically reduces the size of the space which the solver explores while retaining soundness and completeness. CDCL is particularly essential for efficiently resolving UNSAT instances, which require establishing a universal (*all models do not satisfy*), rather than existential (*there exists a satisfying model*), proposition over the search tree.

For ppSAT solving to reach its potential will almost certainly require supporting CDCL. However, this is a challenging task. As an immediate issue, CDCL only learns clauses when backtracking happens. Continuing to hide those occurrences would require a deterministic schedule for adding clauses. Though a simple approach is to add one clause per giant step while padding out with tautologies, every increase to the size of the formula makes every ensuing giant step more expensive. An alternative approach might be to add clauses less frequently, perhaps keeping the largest learned in the interim and discarding the rest. Exploring this frontier will be critical to use of CDCL within a ppSAT solver.

Additionally, the CDCL learning process itself would need to be made oblivious. Usually it is understood as the building of an implication graph for which a suitable (and not necessarily minimal) cut produces the assignments to negate. Though this process may be rendered as a sequence of resolution operations potentially amenable to oblivious formulation [7], doing so without undue overhead may require care.

4.4 Complexity

The overall circuit complexity of our protocol is $O(S \times C)$, where S is the total number of giant steps and C is the cost of each one. The round complexity of our protocol is $O(S)$, as every giant step is a constant-round 2PC of a single circuit. The value of S depends on $\tau_{\lambda,n,m}$, on the number of steps necessary to resolve ϕ, and on whether an exact-time, time-bound, or noisy-time (Appendix D) solver is used.

Our value for C is $O(mn + n \log n)$, though it may vary based on the details of the ADS instantiation and the heuristics. The circuit complexities of UNITSEARCH, CHECK, and PROPAGATION are each $O(mn)$, while that of BACKTRACK is $O(mn + n \log n)$ with the logarithmic term arising from the oblivious stack [60]. UNITSEARCH, CHECK, and PROPAGATION each require $O(1)$ linear scans over the m clauses; during each scan the procedures apply the various ADS operations, each of which require $O(1)$ or $O(n)$ time in our $\beta \approx n$ instantiation. BACKTRACK consists of a pop operation from the oblivious stack of a (partial) model represented in $O(n)$ bits, taking $O(n \log n)$ time, followed by the application of that model to recover the formula state in time $O(mn)$. Finally, DECISION has complexity $O(H + n \log n)$, where H is the complexity of the chosen heuristic and $n \log n$ is the complexity of the oblivious stack push operation. The DLIS and RAND heuristics have $H = O(mn)$. The complexity of Weighted-RAND also requires $H = O(mn)$ so long as the weight of a literal is its frequency in the formula. Other heuristics could have worse (or better) asymptotic cost.

As discussed in §4.1, a RAM-based secure computation solution would reduce the cost of accessing n bits from $O(n)$ to $O(\log^2 n)$. This would, *e.g.*, reduce the cost of ADS operations where we must touch a given literal at every clause, such as $C.\mathsf{contain}(\ell)$, from $O(mn)$ to $O(m \log^2 n)$. With sufficient refining of the data structures, heuristics, and composition of the subroutines these improvements may improve the asymptotic runtime of C in total. However, as noted we can project the protocol efficiency would not be concretely superior at present until at least $n \geq 2^{17}$ [16], with the true crossover depending in part on network conditions as ORAM also requires logarithmic, instead of constant, rounds. We leave the potential of ORAM to future work once algorithm enhancements, like CDCL, make it relevant.

4.5 Obliviousness and Security

At its core our solver raises DPLL into a secure computation using oblivious algorithms and supporting data structures. For the general purpose ADS instantiation (when $\beta \approx n$), our fundamental design choices were to represent both literals and clauses as binary vectors and to manage the decision tree with an oblivious stack to permit backtracking [60]. Using these data structures, the standard DPLL subroutines (such as PROPAGATION or BACKTRACK) can be implemented with linear scans over vectors of fixed public size, oblivious multiplexing of vectors, and push and pop of vectors to and from the oblivious stack. Hence, each of them is data-oblivious.

These individual subroutines must be combined in a manner that maintains data-obliviousness, which is why we adapt DPLL to enforce so-called giant steps (cf. § 4.2). A giant step executes these several small steps in a deterministic order, which are then consolidated through multiplexing. The use of giant steps may result in redundant and ultimately discarded operations, but are necessary to hide the (usually data-dependent) DPLL step being applied. Obliviousness must also be enforced for the decision heuristics. Hence our careful choice of three standard DPLL heuristics amenable to formulation using the same linear scanning and multiplexing techniques: DLIS, RAND, and Weighted-RAND (cf. § 4.3).

A security argument follows from the data-oblivious nature of our solver, the security of two-party computation, and standard composition results [10]. A simulator Sim_{1-b} invokes the simulators for the fixed sequence of circuits, and halts according to τ or $\tau_{\lambda,n,m}$ by injecting s and (optionally) $\mathcal{M}$ into the final output. We refer to [37, 38] for discussion of the proof techniques underlying this sketched argument.

5 Evaluation

Testbed. We implemented our solver using the semi-honest 2PC library of the EMP-toolkit [59]. For managing the oblivious stack we adopted an existing reimplementation of the circuits of Zahur *et al.* [62]. All evaluations were run on a machine with 8GB of RAM and an Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz * 6 processor, with network bandwidth up to 10 Gbps.

Experiment Design. We first measured (§5.1) the performance of a single giant step – in terms of both the number of gates and the runtime of each subroutine – over formulas of various sizes. These evaluations verified our analysis of the asymptotic complexity of ppSAT and its most critical bot-

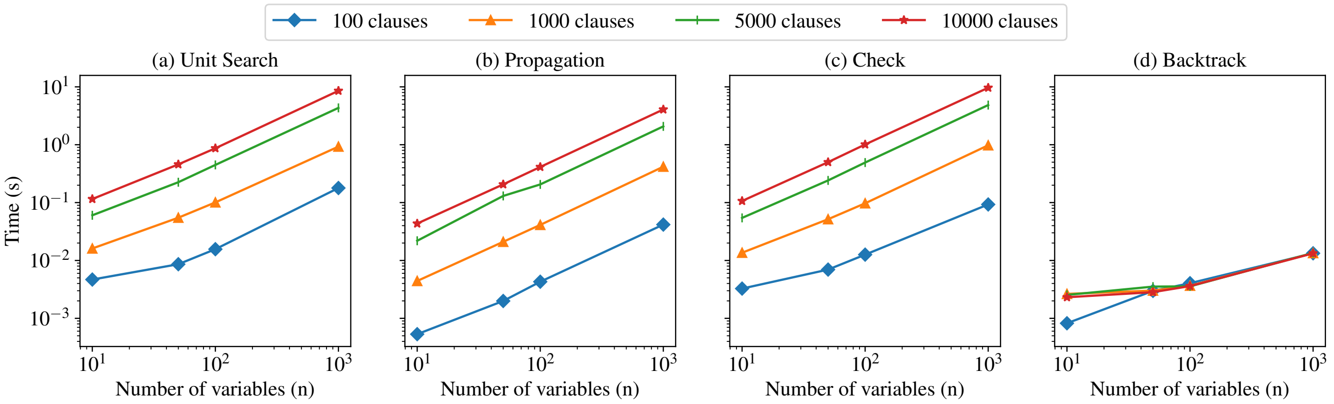

Figure 2: **Subroutine time for our ADS instantiation when $\beta \approx n$.** The runtimes of all four subroutines show linear growth as the number of variables rises. The runtimes of UNITSEARCH, PROPAGATION and CHECK also increase with the number of clauses. Due to our optimized implementation, the runtime of DECISION is independent of the number clauses.

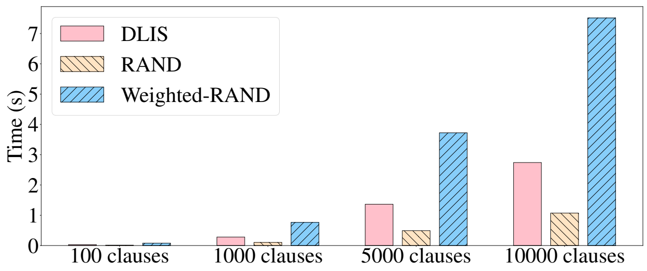

Figure 3: **Time for heuristics when $n = 100$.** The runtime of each heuristic grows with an increase in the number of clauses.

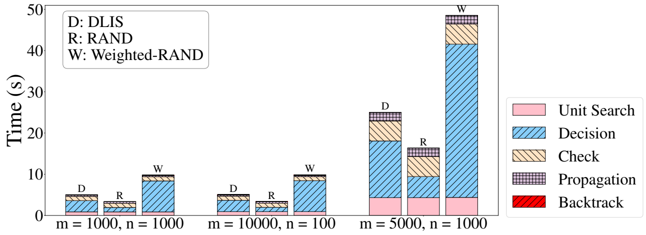

Figure 4: **Time for one giant step varying n, m, and the heuristic.** The DECISION routine dominates the performance of each giant step. The runtime of BACKTRACK is almost negligible in comparison to the other subroutines, and therefore is not visible in the figure.

tlenecks. We then benchmarked (§5.2) our ppSAT solver by testing what proportion of instances it was able to solve (up to a timeout). Finally, we compared (§5.3) the performance of our ppSAT solver against two plaintext solvers: our oblivious ppSAT algorithm executed without cryptographic primitives or communication, and the state-of-the-art Kissat solver [2].

Instance Generation. For measuring the cost of a single giant step the instances were generated randomly. Due to the oblivious nature of the algorithm this has no bearing on the evaluation. For the full evaluation benchmarks, we projected the cost of our solver on small haplotype satifiability instances drawn from the dataset of [42] which was used in the original SHIPs papers [40, 41]. Our instances have parameters of either (i) $|G| \in [1..8]$ and $r = 2|G|$, intended to evaluate the effect of instance size; or (ii) $|G| = 3$ and $r \in [3..6]$, so as to evaluate the effect of instance hardness and (un)satisfiability. We list the resultant formula sizes in terms of n and m in Table 1. Although these databases are smaller than modern HIPP benchmarks, important medical research that motivated early work in computational haplotype inference occurred over datasets where $|G| \approx 10$ [36, 52].

$\vert G\vert$	#var × #clause (≈)		$\vert G\vert$	#var × #clause (≈)	
1	60 × 170		2	150 × 700	
3	$r = 3$	150 ×750	3	$r = 5$	200 × 1200
	$r = 4$	180 × 1000		$r = 6$	250 × 1400
4	350 × 2600		5	400 × 4000	
6	600 × 6000		7	800 × 8000	
8	900 × 10000		—	—	

Table 1: **The size of the formulas for our benchmarks.**

5.1 Micro Benchmarks for Single Giant Steps

We measured the time consumption and number of gates of UNITSEARCH, DECISION, CHECK, and PROPAGATION for each combination of $n \in \{10, 50, 100, 1000\}$ and $m \in \{100, 1000, 5000, 10000\}$, which covers the typical size of instances in older benchmarks such as [30].

Unit Search, Propagation, and Check: The first three rows of Table 2 show the number of gates in UNITSEARCH, PROPAGATION and CHECK for formulas of different sizes. The number of gates increases linearly with both n and m, which is consistent with our asymptotic analysis.

Figures 2(a-c) graph the execution time of UNITSEARCH, CHECK and PROPAGATION. The observed time appears linear

#var × #clause		$100 \times 5K$	$100 \times 10K$	$1K \times 10K$
Unit Search		10	20	200
Propagation		6	12	120
Check		8	16	160
Backtrack		0.02	0.02	0.22
Decision	D	30	60	600
	R	12	24	240
	W	88	175	1740

Table 2: **The approximate number of gates for each subroutine.** The units are in **millions**. D, R, and W refer to the DLIS, RAND, and Weighted-RAND heuristics respectively.

in the number of variables and clauses, though the growth in the latter decreases, likely due to amortization of general overhead. This is as expected, since UNITSEARCH, CHECK and PROPAGATION all run in $O(mn)$ time. The evaluation shows that even for larger instances with $n \approx 1000$ and $m \approx 10000$ these routines cost less than 10 seconds each.

Backtrack: Figure 2(d) shows the time per BACKTRACK execution, which reflects the fourth row of Table 2 in showing that the number of gates increases linearly with n, but is independent of m. Due to an optimization in our implementation the cost of backtracking only depends on the number of variables: we store just the current model (including isAlive) in the oblivious stack, and then recover the formula state within the next multiplexer. As models are just $O(n)$ bits this is independent of the number of clauses. Due to this efficient oblivious stack design the BACKTRACK time for an instance where $n = 1000$ and $m = 10000$ takes only ≈ 0.01s.

Decision: The last row of Table 2 presents the number of gates for DECISION, when using our different heuristics from §4.3. The figure shows DECISION is the most expensive component of ppSAT. While each heuristic linearly scales up in $O(mn)$ time, RAND takes the fewest concrete gates. Figure 3 compares the experimental runtimes when $n = 100$ and with various clause sizes. Again, the observed growth for each heuristic is as expected linear in the number of clauses. The Weighted-RAND heuristic is the most expensive at almost twice the cost of DLIS – likely as it combines RVS with frequency counting. The simpler RAND is cheapest at about only half the time of DLIS.

Giant Step: Figure 4 displays the observed time for a full giant step across a variety of choices for n, m, and heuristic. For instances of the same size the fraction that each component takes remains stable, as expected. For instances of size typical for old benchmarks ($n \approx 100, m \approx 10000$) the time cost is roughly 3s, 5s, and 10s with RAND, DLIS and Weighted-RAND respectively.

5.2 Solving Benchmarks

To evaluate the performance of our solver we first measured the total number of giant steps S for our instances. Although our solver implementation is complete, as cryptographic operations do not affect S to save time we gathered this data with all cryptographic operations removed. We then used the methodology of the micro benchmarks to get a timing C for a single 2PC giant step for those instances, from which the total runtime can be projected as $S \times C$. We also ran the complete ppSAT solver over an UNSAT formula of 1000 variables and 1000 clauses, which took 3019.7 seconds and 532 communication rounds. The projected time was 2993.8 seconds, differing from the real run time by only 0.8%.

We benchmarked our solver using instances we reduced in size from the haplotype inference dataset of [42], specifically the 100*kb* genotype data. We used 232 instances in total to benchmark our solver, varying over $|G|$ and r as previously described. When $r = 2|G|$ the formulas are necessarily satisfiable, while the remaining are mostly unsatisfiable. Both Figure 5 and Figure 6 depict the proportion of the instances solved within a particular time, *i.e.*, a point (x, y) indicates that y proportion of the instances are projected to be solved within x seconds. We set the timeout to 200k seconds (≈ 2.3 days).

For the first trial (Figure 5) the instances vary in $|G|$ while r is fixed to be $2|G|$. All three heuristics can solve most formulas before the timeout for $|G| \leq 3$, but vary in performance when the formulas get larger. For those smaller formulas DLIS outperforms RAND and Weighted-RAND. However, when $|G| > 3$, Weighted-RAND outperforms the other two, and when $|G| = 8$ it is the only heuristic with which the solver can successfully solve any benchmarks. This result is expected and reasonable: though expensive per giant step, it is the only one of the three to combine randomness with insight into the formula structure (through the weighting).

In the second trial (Figure 6) we evaluated the performance of our solver for various r with fixed $|G| = 3$. When $r < 2|G|$ the formula can (i) be unsatisfiable; or (ii) remain satisfiable but potentially be more difficult, as it requires some haplotypes to explain more than one genotype. The solver can handle over 70% instances before the timeout for all heuristics and almost all r, though the RAND heuristic leads to only 30% success when $r = 5$. Despite the larger formulas the solver is more successful for $r = 6$ than for $r = 5$, likely due to the greater solution freedom explained by (i) and (ii).

5.3 Comparison to Plaintext Solvers

Finally, we compared our ppSAT solver against itself when run in the plaintext. We wrote our plaintext solver in Python by implementing our pseudocode in the natural way, *i.e.*, every garbled circuit was replaced with standard RAM model operations, and a non-oblivious stack was used. Table 3 shows the results and so the overhead brought by communication and 2PC. We also compared ppSAT with the state-of-the-art Kissat SAT solver [2]. For our 232 benchmarks Kissat can solve 231 of the instances within 0.02s, and the last within 1s. For brevity we omit the raw data from this experiment.

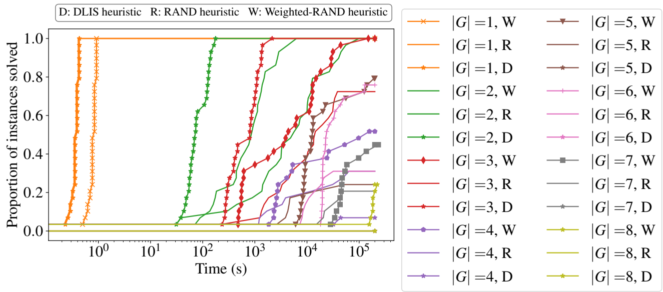

Figure 5: **Haplotype benchmarks for when $|G| \in [1..8]$ and $r = 2|G|$ with timeout of 200k seconds.** With all heuristics the solver is successful on small formulas, *e.g.*, $|G| \leq 2$, for which DLIS outperforms the randomized heuristics. Weighted-RAND becomes the most effective for larger databases.

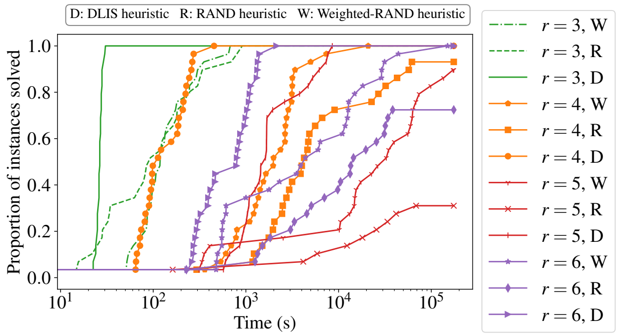

Figure 6: **Haplotype benchmarks for $r \in \{3,4,5,6\}$ when $|G| = 3$ with timeout of 200k seconds.** Our solver resolves over 80% of benchmarks before timing out, except for RAND when $r = 5$.

6 Conclusion

The field of SAT solving has seen a superb (and continuing) developmental arc since the publication of DPLL almost 60 years ago. Given its centrality to computing and the importance of data privacy to modern technology and society, efficient privacy-preserving SAT solving would likely be a versatile and powerful tool for research and practice. In this paper we established the core security definitions, oblivious DPLL design, and private decision heuristics necessary to implement a ppSAT solver capable of resolving small but practical instances. Of perhaps greater importance than our empirical results is the basis this lays for future work towards more efficient and effective ppSAT solvers, which we might hope will retrace the developmental arc of SAT solving itself.

Limitations & Future Directions. The centrality of CDCL to modern SAT solving makes its reformulation for ppSAT, as discussed in §4.3, the most important direction for future work. The greatest limitation of the original DPLL algorithm – and so also of our oblivious adaption of it – is its inability to effectively learn from failed branches of its search. CDCL is the dominant enhancement of DPLL for rectifing this shortcoming [20], and it is hard to imagine a path to general practicality for ppSAT solving that does not also rely upon it, especially for UNSAT instances.

#var × #clause	50 × 10K	100 × 10K	1K × 10K
RAND	3.4×	5.1×	47.0×
Weighted-RAND	8.3×	11.0×	165×
DLIS	4.6×	6.4×	136.8×

Table 3: **Slowdown of ppSAT compared with it in the plaintext.** In the plaintext means all data and operations are public during the computation.

Pruning the search tree is not the only tactic, however, for beating back the combinatorial explosion of DPLL. Modern SAT solvers rely heavily on "making their own luck" for searching what remains through intelligent decision heuristics. Our DLIS, RAND, and Weighted-RAND heuristics are limited by the standards of modern solvers [45], but have the benefit for us of being naturally implementable within Boolean circuits. Adapting or developing a fresh suite of effective heuristics suitable for oblivious computation – perhaps even using mixed-mode MPC (*i.e.*, with both Boolean and arithmetic circuit primitives) – is another major future direction. Decision heuristics also provide a particularly fertile ground for further collaboration between the cryptography and formal methods communities, as they will require reconciling the algorithmic techniques of each. Together, adapting CDCL and developing suitable decision heuristics are the foremost steps to generally practical ppSAT solving.

The last two directions for future work we emphasize are discussed more comprehensively in Appendix D. The first is to begin to understand the practical meaning of any privacy loss permitted by our security definitions. There is limited prior work on characterizing information leakage from MPC for general computations, with that of Mardziel *et al.* [43] the only known to the authors. At present we cannot in any meaningful sense explain what a party loses in privacy by, *e.g.*, setting a specific $\tau_{\lambda,n,m}$ based on an economic analysis of projected runtime using our micro benchmarks methodology, or choosing an exact-time-revealing solver over its time-bound-revealing cousin. Given the rich and complex encoding of information within the structure of SAT formulas, exploring how ppSAT solvers should leak information (which may be especially important to integrating CDCL) is likely necessary for widespread confidence in their future practical deployment. Finally, in the appendix we also discuss a collection of preprocessing optimizations which trade off efficiency against privacy. As the core algorithms of ppSAT solving develop and mature, expanding the suite of such techniques may further encourage the development of practical tooling.

Acknowledgments

The authors thank Peter Chan for making available his oblivious stack implementation. This work was supported by the ONR through an NDSEG Fellowship and under Grant N00014-17-1-2787, by NSF awards CCF-2106845, CCF-2131476, CNS-2016240, and CNS-1565208, as well as by research awards from Facebook, Google and PlatON.

References

[1] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal Oblivious RAM. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '20)*, 2020.

[2] Tomáš Balyo, Nils Froleyks, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions. 2021.

[3] Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. Counting Models Using Connected Components. In *National Conference on Artificial Intelligence (AAAI '00)*, 2000.

[4] Roberto J. Bayardo Jr and Robert C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *National Conference on Artificial Intelligence (AAAI '97)*, 1997.

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*.

[6] Armin Biere. Lingeling Essentials, A Tutorial on Design and Implementation Aspects of the the SAT Solver Lingeling. *Pragmatics of SAT (PoS@SAT '14)*, 2014.

[7] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of Satisfiability*. IOS press, 2009.

[8] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-Oblivious Graph Algorithms for Secure Computation and Outsourcing. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS' 13)*, 2013.

[9] Paul Bunn and Rafail Ostrovsky. Secure Two-Party k-means Clustering. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*, 2007.

[10] Ran Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1), January 2000.

[11] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC '71)*, 1971.

[12] Mark J. Daly, John D. Rioux, Stephen F. Schaffner, Thomas J. Hudson, and Eric S. Lander. High-Resolution Haplotype Structure in the Human Genome. *Nature Genetics*, (2), 2001.

[13] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM (CACM)*, (7), 1962.

[14] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM (JACM)*, (3), 1960.

[15] Damien Desfontaines and Balázs Pejó. Sok: Differential Privacies. *Proceedings on Privacy Enhancing Technologies (PoPETS '20)*, (2), 2020.

[16] Jack Doerner and Abhi Shelat. Scaling ORAM for Secure Computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, 2017.

[17] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography Conference (TCC '06)*. Springer, 2006.

[18] Cynthia Dwork and Aaron Roth. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science*, (3-4), 2014.

[19] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient Private Matching and Set Intersection. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '04)*. Springer, 2004.

[20] Vijay Ganesh and Moshe Vardi. On The Unreasonable Effectiveness of SAT Solvers. In Tim Roughgarden, editor, *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press, 2020.

[21] 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 *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, 2012.

[22] Ana Graça, Inês Lynce, Joao Marques-Silva, and Arlindo L. Oliveira. Haplotype Inference by Pure Parsimony: a Survey. *Journal of Computational Biology*, (8), 2010.

[23] Adam Groce, Peter Rindal, and Mike Rosulek. Cheaper Private Set Intersection via Differentially Private Leakage. *Proceedings on Privacy Enhancing Technologies (PoPETS '19)*, (3), 2019.

[24] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program Analysis as Constraint Solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, 2008.

[25] Dan Gusfield. Haplotype Inference by Pure Parsimony. In *Annual Symposium on Combinatorial Pattern Matching (CPM '03)*. Springer, 2003.

[26] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, (4), 2010.

[27] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. Sok: General Purpose Compilers for Secure Multi-Party Computation. In *2019 IEEE Symposium on Security and Privacy (S&P '19)*, pages 1220–1237. IEEE, 2019.

[28] Xi He, Ashwin Machanavajjhala, Cheryl Flynn, and Divesh Srivastava. Composing Differential Privacy and Secure Computation: A Case Study on Scaling Private Record Linkage. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, 2017.

[29] Nils Homer, Szabolcs Szelinger, Margot Redman, David Duggan, Waibhav Tembe, Jill Muehling, John V. Pearson, Dietrich A. Stephan, Stanley F. Nelson, and David W. Craig. Resolving Individuals Contributing Trace Amounts of DNA to Highly Complex Mixtures using High-Density SNP Genotyping Microarrays. *PLoS Genet*, (8), 2008.

[30] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. *International Conference on Theory and Applications of Satisfiability Testing (SAT '00)*, 2000.

[31] Russell Impagliazzo and Ramamohan Paturi. On the Complexity of k-SAT. *Journal of Computer and System Sciences*, (2), 2001.

[32] Richard M. Karp. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*. Springer, 1972.

[33] Lea Kissner and Dawn Song. Privacy-Preserving Set Operations. In *Annual International Cryptology Conference (CRYPTO '05)*. Springer, 2005.

[34] Giuseppe Lancia, Maria Cristina Pinotti, and Romeo Rizzi. Haplotyping Populations by Pure Parsimony: Complexity of Exact and Approximation Algorithms. *INFORMS Journal on Computing*, (4), 2004.

[35] K. Rustan M. Leino. A SAT Characterization of Boolean-Program Correctness. In *International SPIN Workshop on Model Checking of Software (SPIN '03)*. Springer, 2003.

[36] Zhenping Li, Wenfeng Zhou, Xiang-Sun Zhang, and Luonan Chen. A Parsimonious Tree-Grow Method for Haplotype Inference. *Bioinformatics*, (17), 2005.

[37] Yehuda Lindell. How to Simulate It – a Tutorial on the Simulation Proof Technique. *Tutorials on the Foundations of Cryptography*, 2017.

[38] Yehuda Lindell and Benny Pinkas. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology*, (2), 2009.

[39] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, and George Varghese. Network Verification in the Light of Program Verification. Technical report, Microsoft Research, 2013.

[40] Inês Lynce and Joao Marques-Silva. Efficient Haplotype Inference with Boolean Satisfiability. In *National Conference on Artificial Intelligence (AAAI '06)*. AAAI Press, 2006.

[41] Inês Lynce and João Marques-Silva. SAT in Bioinformatics: Making the Case with Haplotype Inference. In *International Conference on Theory and Applications of Satisfiability Testing (SAT '06)*. Springer, 2006.

[42] Jonathan Marchini, David Cutler, Nick Patterson, Matthew Stephens, Eleazar Eskin, Eran Halperin, Shin Lin, Zhaohui S Qin, Heather M. Munro, Gonçalo R. Abecasis, Peter Donnelly, and International HapMap Consortium. A Comparison of Phasing Algorithms for Trios and Unrelated Individuals. *The American Journal of Human Genetics*, (3), 2006.

[43] Piotr Mardziel, Michael Hicks, Jonathan Katz, and Mudhakar Srivatsa. Knowledge-oriented Secure Multiparty Computation. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security (PLAS '12)*, pages 1–12, 2012.

[44] Filip Marić. Formalization and Implementation of Modern SAT Solvers. *Journal of Automated Reasoning*, (1), 2009.

[45] Joao Marques-Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *Portuguese Conference on Artificial Intelligence*. Springer, 1999.

[46] Ruben Martins, Vasco Manquinho, and Inês Lynce. An Overview of Parallel SAT Solving. *Constraints*, (3), 2012.

[47] Kenneth L. McMillan. Interpolation and SAT-based Model Checking. In *International Conference on Computer Aided Verification (CAV '03)*. Springer, 2003.

[48] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, 2001.

[49] Paul Ohm. Broken Promises of Privacy: Responding to the Surprising Failure of Anonymization. *UCLA L. Rev.*, 2009.

[50] Nila Patil, Anthony J. Berno, David A. Hinds, Wade A. Barrett, Jigna M. Doshi, Coleen R. Hacker, Curtis R. Kautzer, Danny H. Lee, Claire Marjoribanks, David P. McDonough, et al. Blocks of Limited Haplotype Diversity Revealed by High-Resolution Scanning of Human Chromosome 21. *Science*, (5547), 2001.

[51] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster Private Set Intersection Based on OT Extension. In *USENIX Security Symposium (USENIX Security '14)*, 2014.

[52] Mark J. Rieder, Scott L. Taylor, Andrew G. Clark, and Deborah A. Nickerson. Sequence Variation in the Human Angiotensin Converting Enzyme. *Nature Genetics*, (1), 1999.

[53] Thomas J. Schaefer. The Complexity of Satisfiability Problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing (STOC '78)*, 1978.

[54] J.P. Marques Silva and K.A. Sakallah. GRASP-A New Search Algorithm for Satisfiability. In *Proceedings of*

International Conference on Computer Aided Design (ICCAD '96). IEEE, 1996.

[55] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In *International Conference on Theory and Applications of Satisfiability Testing (SAT '06)*. Springer, 2009.

[56] Niklas Sorensson and Niklas Een. Minisat v1. 13-A SAT Solver with Conflict-Clause Minimization. *SAT*, (53), 2005.

[57] K. Sunder Rajan. *Biocapital: The Constitution of Postgenomic Life*. Duke University Press, 2006.

[58] Rui Wang, Yong Fuga Li, XiaoFeng Wang, Haixu Tang, and Xiaoyong Zhou. Learning Your Identity and Disease from Research Papers: Information Leaks in Genome Wide Association Study. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, 2009.

[59] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-Toolkit: Efficient MultiParty Computation Toolkit. `https://github.com/emp-toolkit`, 2016.

[60] Xiao Wang, Kartik Nayak, Chang Liu, T.H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*, 2014.

[61] Andrew Chi-Chih Yao. How to Generate and Exchange Secrets. In *27th Annual Symposium on Foundations of Computer Science (FOCS '86)*. IEEE, 1986.

[62] Samee Zahur and David Evans. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In *2013 IEEE Symposium on Security and Privacy (S&P '13)*. IEEE, 2013.

A Details of Our Security Definition

Our formal definition of a secure ppSAT solver has four variants depending on when it halts and whether a model is output. For brevity, we only explicitly give the most comprehensive.

Definition A.1 (Two-Party Exact-Time-and-Model-Revealing ppSAT Solver). *Let λ be a security parameter, ϕ_0, ϕ_1, ϕ_{pub} be Boolean propositional formulas over variables $v_1, \ldots, v_n$ such that $m = |\phi_0| + |\phi_1| + |\phi_{pub}|$, and $T(\lambda, n, m) = \tau_{\lambda,n,m}$ be a polynomial. For $b \in \{0, 1\}$, let*

$$v_b^{real} = \mathrm{view}_b^{\Pi}(\phi_b, n, m, \phi_{pub}, \tau_{\lambda,n,m}, P_{1-b}(\phi_{1-b}, n, m, \phi_{pub}, \tau_{\lambda,n,m})),$$

and

$$v_b^{ideal} = \mathrm{view}_b^{\Pi}(\phi_b, n, m, \phi_{pub}, \tau_{\lambda,n,m}, \mathrm{Sim}_{1-b}(\phi_b, n, m, \phi_{pub}, \tau_{\lambda,n,m}, s, \mathcal{M}, \tau)).$$

be the real and ideal views of P_b after executing protocol

$$(s, \mathcal{M}) \leftarrow \Pi(\phi_0 \parallel \phi_1; n, m, \phi_{pub}, \tau_{\lambda,n,m})$$

terminating in $\tau \leq \tau_{\lambda,n,m}$ steps. Then Π is a two-party exact-time-and-model-revealing privacy-preserving SAT solver (2p-etmr-solver) if

1. *$s = 1$ iff $\exists \mathcal{M}'.(\mathcal{M} = \mathcal{M}') \wedge \mathcal{M}' \models \phi$;*
2. *$s = 0$ iff $\nexists \mathcal{M}'.\mathcal{M}' \models \phi$; and*
3. *there exists Sim_{1-b} such that for any probabilistic polynomial-time (PPT) decision algorithm $\mathcal{A}$:*

   $$|\Pr[\mathcal{A}(1^\lambda, v_b^{real}) = 1] - \Pr[\mathcal{A}(1^\lambda, v_b^{ideal}) = 1]| \leq \mathrm{negl}(\lambda)$$

   where $\mathrm{negl}(\lambda)$ is eventually bounded above by the inverse of every polynomial function of λ.

The three other variants of this definition are (i) a time-bound-and-model-revealing solver (*2p-tbmr*-solver), where the simulator is not given τ; (ii) an exact-time-revealing solver (*2p-etr*-solver), where $\mathcal{M}$ is removed from the definition; and (iii) a time-bound-revealing solver (*2p-tbr*-solver) which combines the changes from (i) and (ii). Intuitively, the time-bound-revealing definitions require the protocol to run for $\tau_{\lambda,n,m}$ steps always, while the exact-time-revealing definitions allow halting immediately upon resolution, and require aborting at $\tau_{\lambda,n,m}$ if necessary.

B Instantiating the ADS for $\beta \ll n$

Recall that $\beta \leq n$ is the maximum clause length in an instance ϕ. When it is publicly known that the maximum number of literals appearing in any given clause is small, an alternative approach to binary clausal vectors is to use signed integers to represent both assignments and literals; *i.e.*, $\neg v_j$ and the assignment $v_j = 0$ are each represented by $-j$. A clause $C \in \phi$ is encoded by two vectors, $\mathsf{literals} \in \mathbb{Z}^\beta$ and $\mathsf{active} \in \{0, 1\}^\beta$. The $\mathsf{literals}$ vector is composed of β signed integers which encode the literals and their sign, padded out with zeros as necessary. The active vector encodes whether the i-th literal has been removed from C. As an example, at initialization $(v_1 \vee v_3 \vee \neg v_5)$ with $\beta = 4$ will be encoded as $\mathsf{literals} = [1, 3, -5, 0]$ and $\mathsf{active} = [1, 1, 1, 0]$. A literal or assignment is negated by flipping the sign.

Determining if a clause is unit can be implemented by scanning active and checking if it has a unique non-zero entry. To check membership of an $\ell = j \in C$ the algorithm checks if there exists an i such that $\mathsf{active}[i] = 1$ and $\mathsf{literals}[i] = j$, while removing it is accomplished by setting $\mathsf{active}[i] = 0$ when $\mathsf{literals}[i] = j$.

At rest this instantiation provides more efficient clausal representations so long as $\beta \cdot k < n$ where $k > \log n$ is the bit-length of the integer encoding. In practice, our evaluation of this approach suffered in comparison to that for $\beta \approx n$ due to the reduced efficacy of an implementation optimization for the oblivious stack. Specifically, to reduce the cost of the stack operations our code only stores the current set of assignments (including $\mathsf{isAlive}$) within it, and then reconstructs the formula

Algorithm 7: Clausal Algorithms when $\beta \ll n$

```
Function $C_j$.unit():
    $b_1 \leftarrow 0$; $b_2 \leftarrow 0$
    for $i \leftarrow 1$ to $\beta$ do
        $b_2 \leftarrow (\mathsf{active}[i] \wedge b_1) \vee b_2$
        $b_1 \leftarrow \mathsf{active}[i] \vee b_1$
    return $b_1 \wedge \neg b_2$
Function $C_j$.contain($\ell \in \mathbb{Z}$):
    $b \leftarrow 0$
    for $i \leftarrow 1$ to $\beta$ do
        $b \leftarrow b \vee (\mathsf{active}[i] \wedge \mathsf{literals}[i] = \ell)$
    return $b$
Function $C_j$.remove($a \in \mathbb{Z}$):
    for $i \leftarrow 1$ to $\beta$ do
        $b \leftarrow (\mathsf{literals}[i] = a)$
        $\mathsf{active}[i] = \neg b \wedge \mathsf{active}[i]$
```

Algorithm 8: Unit Search

```
Input: $\phi$
Output: $b \in \{0, 1\}$, $a = (\mathsf{ind}^+, \mathsf{ind}^-)$
$a \leftarrow \perp$; $b \leftarrow 0$ ;
for $j \leftarrow 1$ to $m$ do
    $u_j \leftarrow C_j$.unit();
    if $u_j = 1$ then
        $a \leftarrow C_j$; $b \leftarrow 1$;
return $b$, $a$
```

Algorithm 9: Check

```
Input: $\phi$, $\ell = (\mathsf{ind}^+, \mathsf{ind}^-)$
Output: $b \in \{0, 1, 2\}$
$b_0 \leftarrow \phi$.empty();
$b_1 \leftarrow 0$;
for $j \leftarrow 1$ to $m$ do
    if $C_j$.unit() $\wedge$ $C_j$.contain($\neg\ell$) then
        $b_1 \leftarrow 1$;
if $b_0 = 1$ then
    return 0;
else if $b_1 = 1$ then
    return 1;
else
    return 2;
```

Algorithm 10: Propagation

```
Input: $\phi$, $a = (\mathsf{ind}^+, \mathsf{ind}^-)$
Output: $\phi'$
for $j \leftarrow 1$ to $m$ do
    $b_0 \leftarrow C_j$.contain($a$);
    $b_1 \leftarrow C_j$.contain($\neg a$);
    if $b_0 = 1$ then
        $\phi$.remove($C_j$);
    if $b_1 = 1$ then
        $C_j$.remove($\neg a$);
return $\phi$
```

during a backtrack. The signed integer encoding requires spending a few hundred gates to compare every assignment to every literal, of which there are $\beta \cdot nm$ such comparisons. Resolving this gap is a potential optimization path.

C Subroutine Definitions

We give our formal definitions for the UNITSEARCH, CHECK, and PROPAGATION subroutines (see §4) as Algorithms 8-10.

D Additional Considerations

We raise a few additional considerations worthy of expansion, which also point towards potential future research directions.

Noisy Termination. There are inherent compromises to both exact-time-revealing and time-bound-revealing solvers. For the former, it is not immediate how much information leakage occurs when halting at resolution. In some circumstances it may be significant. For example, if ϕ_0 and ϕ_1 each contain unit clauses (v_i) and $(\neg v_i)$ respectively then the solver will always halt on the first giant step. If the inclusion of these clauses carries privacy implications then this leakage may be unacceptable. Nonetheless, it seems plausible that for many natural instances such runtimes are too coarse a measure to contain information compromising to privacy in practice – especially when using randomized heuristics. As for time-bound-revealing solvers, running for $\tau_{\lambda,n,m}$ steps may be expensive and undesirable when not required for correctness. Always requiring such a high cost could very well limit the economic or social value of the solver.

A potential third way is to not terminate exactly upon resolution, but instead to add calibrated noise to extend the runtime for a manageable but privacy-enhancing number of steps. The theory of differential privacy (DP) [17, 18] would seem to provide an applicable toolkit, and has in fact been integrated with 2PC for the closely related purpose of "noisy load over-estimation" in a line of recent work [23, 28]. The intuitive idea, following He *et al.* [28], is to relax Definition A.1 to allow a bounded difference in the output of the adversary for any two formulas of the same length. However, we cannot directly use their formulation of *output-constrained DP*, since in our case only some of the output (τ) will have added noise on release – both s and (when applicable) $\mathcal{M}$ will be released exactly.

Instead, we formulate a $(\varepsilon_0, \varepsilon_1, \delta_0, \delta_1)$-noisy-time-and-model-revealing ppSAT solver (or $(\varepsilon_0, \varepsilon_1, \delta_0, \delta_1)$-*2p-ntmr-*

solver) by altering Definition A.1 so that (i) instead

$$v^{ideal}_{b,\phi'_{1-b}} = \text{view}^{\Pi}_{b}(\phi_b, n, m, \phi_{pub}, \tau_{\lambda,n,m}, \varepsilon_b, \delta_b,$$
$$\text{Sim}_{1-b}(\phi_b, \phi'_{1-b}, n, m, \phi_{pub}, \tau_{\lambda,n,m}, \varepsilon_{1-b}, \delta_{1-b}, s, \mathcal{M}))$$

for $\varepsilon_b, \varepsilon_{1-b} > 0$ and $0 \leq \delta_b, \delta_{1-b} < 1$, and (ii) requiring that there exist a Sim_{1-b} such that:

$$\Pr[\mathcal{A}(1^\lambda, v^{real}_b) = 1] \leq e^{\varepsilon_{1-b}} \cdot \Pr[\mathcal{A}(1^\lambda, v^{ideal}_{b,\phi'_{1-b}}) = 1] + \delta_{1-b}$$

for all ϕ'_{1-b} such that $|\phi_{1-b}| = |\phi'_{1-b}|$. Intuitively the simulator no longer has τ, and so instead must internally execute the ppSAT solver over $\phi' = \phi_b \wedge \phi'_{1-b} \wedge \phi_{pub}$ to determine a resolution time ρ_S, before adding noise to determine a τ_S to halt at. As the real world stopping time τ_R is also a noisy version of the true resolution time ρ_R, suitable noise will allow the simulator to meet the requirement for $(\varepsilon_{1-b}, \delta_{1-b})$-indistinguishability.

Unfortunately the instability of SAT instances makes this guarantee difficult to practically realize. With this definition we are viewing the ppSAT solver as a mechanism which on input a "database" in the form of ϕ noisily outputs the resultant runtime $\tau_\phi \in [1..\tau_{\lambda,n,m}]$. The amount of noise required depends on the sensitivity $\Delta\tau = \max_{\phi,\phi'} \max_{\tau_\phi,\tau_{\phi'}} |\tau_\phi - \tau_{\phi'}|$ for all pairs ϕ, ϕ' where $|\phi_{1-b}| = |\phi'_{1-b}|$ and $\phi_{pub} = \phi'_{pub}$. However, $\Delta\tau$ can often be a significant fraction of $\tau_{\lambda,n,m}$ as it is taken over all ϕ' of a certain length, including, *e.g.*, cryptographic instances [20]. This is far larger of a sensitivity than DP mechanisms naturally work well with. For example, applying the load overestimating techniques from [23, 28] would lead to increasing the runtime by $\approx 40 \cdot \Delta\tau/\varepsilon$ giant steps on average [23], while we ideally want $\varepsilon < 1$ and certainly desire it to be small [18].

When applying DP outside its origin in private statistical data analysis, a common technique is to reduce the anonymity set by restricting the sensitivity to some other definition of pairs of "adjacent" instances [15]. However, it is unclear how to do this for SAT instances in a reasonable way. Characterizing what makes SAT instances natural is a deeply rich and complex question with numerous mathematical and empirical notions in the literature – clause density, treewidth, backdoors, modularity, etc. [7, 20]. There is no immediately apparent way to decide which formulas to include in a definition of adjacency based on these metrics, nor how to prove a usefully reduced sensitivity from them. Alternatively, in some settings empirical analysis might show that distributions of runtimes are nicely clustered for both SAT and UNSAT instances, allowing noise calibrated to smaller sensitivities justified on those statistics. But even then the sensitivity may very well require impractical noise.

We mostly leave these questions open. One potential direction might be to make a leap in logic and characterize the output of the mechanism as the proportion $\tau/\tau_{\lambda,n,m}$. The exponential mechanism [18] could then be used with bucketing of runtimes and an MPC outcome selection similar to the approach used in Algorithm 6. Though this is not justified on first principles it may be defensible. In private data analysis the sensitivity of a uniquely identifying query (*e.g.*, "count 'John Doe with UID: 1234' entries in the database") exactly captures the worst-case information leakage. Since the leakage from SAT runtimes is arguably much coarser, the proportion of the available time used may be a more natural interpretation of leakage than the actual number of giant steps. But such an approach would compromise the firm foundations of the DP guarantee.

Preprocessing Optimizations. Continuing along the lines of trading off efficiency for leakage, we briefly raise a few potential preprocessing optimizations. In general, developing a suite of similar such techniques, as well as an understanding of the tradeoffs they bring, may be a rich avenue for future work.

1. Though private set disjointedness is likely unreasonable over the (often massive) set of valid models, it could be used to find unit clause conflicts before initializing the full SAT solver. For example, suppose ϕ_1 has forced variables $\text{pos}_0 = \{v_1, v_3\}$, and $\text{neg}_0 = \{v_{17}\}$, *i.e.*, $v_1 = 1, v_3 = 1, v_{17} = 0$ must all necessarily be assigned. If ϕ_1 has forced $\text{pos}_1 = \{v_1, v_9\}$ and $\text{neg}_1 = \{v_3\}$, then the P_i could determine that $|\text{pos}_0 \cap \text{neg}_1| + |\text{neg}_0 \cap \text{pos}_1| > 0$ and output UNSAT immediately. The tradeoff would be to leak information about the composition of these sets.

2. Another potential technique would be to allow parties to provide "hints", *i.e.*, partial models which satisfy (part of) their formula. In particular, each party could provide a set of assignments which resolve especially tricky structures within their input. By loading all of these hints into another oblivious stack they could each be explored from, ultimately falling back on an empty model if none are successful.

3. A final idea applies when ϕ can be split into subsets of clauses all of which are independent from each other in terms of the variables they reference. For example, $\phi = (v_1) \wedge (v_1 \vee v_2) \wedge (v_3 \vee v_4)$ may be split into $\phi_a = (v_1) \wedge (v_1 \vee v_2)$ and $\phi_b = (v_3 \vee v_4)$. If the P_i are willing to leak the variable inclusions in these subinstances it would allow running them independently, potentially reducing costs as

$$(2^{n_1} \log n_1) \cdots (2^{n_k} \log n_k) \leq$$
$$(2^{n_1} + \cdots + 2^{n_k}) \log(n_1 + \cdots + n_k).$$

   for k subinstances each of n_i variables for $i \in [k]$. Privately finding these subinstances could be done through an oblivious breadth-first search for strongly connected components over the adjacency graph of ϕ, adapting from [8].