Extending CDCL to Disjunctions of Parity Equations

Paul Beame
University of Washington, Seattle, WA, USA

Glenn Sun
University of Washington, Seattle, WA, USA

Abstract

Because CDCL produces proofs in the Resolution proof system, problems provably hard for Resolution are also provably hard for CDCL. Exponentially shorter proofs can sometimes be found using stronger proof systems such as Res($\oplus$), a generalization of Resolution to XNF formulas, whose constraints are disjunctions of parity equations ("linear clauses") such as $(x \oplus y) \vee \neg(y \oplus z)$. While some modern solvers like CryptoMiniSAT reason on Boolean clauses with separate parity equations, reasoning about more general linear clauses is less explored.

We present CDCL($\oplus$), a generalization of CDCL to XNF formulas, and prove a bidirectional connection with Res($\oplus$): CDCL($\oplus$) not only produces Res($\oplus$) proofs, but also polynomially simulates Res($\oplus$) given nondeterministic decisions and restarts, mirroring the classical relationship between CDCL and Resolution. Our key technical tool is a new set of inference rules for Res($\oplus$) that helps us translate Resolution-based subroutines such as 1-UIP clause learning. Altogether, CDCL($\oplus$)'s parity reasoning includes branching on arbitrary parity equations, linear-algebraic reasoning during unit propagation, and learning linear clauses through conflict analysis.

We provide a proof-of-concept implementation of CDCL($\oplus$) called Xorcle, which includes adaptations of existing CDCL heuristics to XNF formulas and an extension of LRUP proof logging that we call LRUP($\oplus$). On a selected suite of benchmarks focusing on native XNF formulas, Xorcle outperforms existing solvers such as Kissat and CryptoMiniSAT. Additionally, on Tseitin formulas written in CNF, even without preprocessing, Xorcle's running time appears to scale nearly polynomially.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Proof complexity

Keywords and phrases SAT, CDCL, Proof Complexity, Parity Reasoning

Digital Object Identifier 10.4230/LIPIcs.SAT.2026.5

Related Version *Full Version*: `https://arxiv.org/abs/2605.15002` [6]

Supplementary Material *Software (Source Code)*: `https://github.com/glenn-sun/xorcle` [51] archived at `swh:1:snp:4db00879b68ea0da4cb4079628e526b4c21121bc`

Funding This work was supported by NSF grant CCF-2422205.

Acknowledgements Parts of the source code were written with OpenAI Codex. All source code has been manually reviewed and tested.

1 Introduction

Although conflict-driven clause learning (CDCL) is the dominant method today for SAT solving, it also has known limitations. Because CDCL produces proofs of unsatisfiability in the Resolution proof system, all problems with exponential lower bounds for Resolution proof size must also be hard for CDCL [5]. These limitations have led some solvers to incorporate ideas and fragments of stronger proof systems: For example, pseudo-Boolean solvers [10, 45, 55] use ideas from Cutting Planes proofs [28, 15, 17], bounded variable addition [39] includes a fragment of Extended Resolution [53], and CNF-XOR solvers [49] include some parity reasoning present in Res($\oplus$) ("Res-parity") [32, 33].

In contrast to these situations where solvers apply some aspects of the corresponding proof system, the relationship between classical CDCL and Resolution is *bidirectional*: not only does CDCL produce Resolution proofs, but with the right choice of heuristic decisions, any Resolution proof can also be turned into a CDCL execution [44]. More precisely, when we view the non-deterministic versions of CDCL solvers as proof systems in their own right, they can polynomially simulate (i.e., *p-simulate* [16]) Resolution proofs.

In this work, we define a CDCL-style algorithm that not only uses Res($\oplus$) reasoning, but also *p*-simulates all of Res($\oplus$), providing the same precise understanding that we have for classical CDCL. Res($\oplus$) handles formulas with constraints involving XORs, which occur in a variety of areas including cryptanalysis and model counting [3, 26]. A widely studied special case of Res($\oplus$)-based solving, known as *CNF-XOR solving* [4, 13, 34, 35], reasons with standard Boolean clauses in conjunction with *parity equations*: expressions involving only $\oplus$ and negation, such as $\neg(x \oplus y \oplus z)$, which can be written in equation form as $[x \oplus y \oplus z = 0]$. CryptoMiniSAT [49, 46, 48, 47] is a particularly important CNF-XOR solver, which both performs parity reasoning such as Gaussian elimination on parity equations and also detects short parity equations encoded in Boolean clauses.

Although CNF-XOR solvers use Res($\oplus$) reasoning, Res($\oplus$) is more expressive in general. In Res($\oplus$), instead of separate Boolean clauses and parity equations, there is only one kind of object that generalizes both: a *linear clause* is a disjunction of parity equations, such as $(x \oplus y) \vee \neg(y \oplus z)$. A conjunction of linear clauses is sometimes called an XOR-OR-AND normal form (XNF) formula [1]. Not only is XNF more expressive than CNF-XOR, but Res($\oplus$) can also succinctly prove some problems, such as the bijective pigeonhole principle [32, 29], that have no known polynomial-length proofs in CNF-XOR systems.

While XNF-solving has received less attention than CNF and CNF-XOR solving, some proposed algorithms for XNF-solving exist. In their original paper defining Res($\oplus$) [32], Itsykson and Sokolov outlined a backtracking search procedure and proved that it is equivalent to the tree-like fragment of Res($\oplus$). Horáček [30] defined an algebraic version of unit propagation in a proof system denoted SRES, inspired by Gröbner basis algorithms and equivalent to Res($\oplus$), yielding an analogue of DPLL when combined with the algorithm of [32]. Two other proposed algorithms use different methods entirely: (1) a brute-force search through possible ways to apply SRES inference rules [31], and (2) a graph-based approach to solve 2-XNF formulas (the XNF analog of 2-CNF formulas) called 2-Xornado [1].

Most recently, in work that is concurrent and independent of ours, Danner and Kreuzer [19, 20] show how to use clause learning with XNF formulas. They also implement a restricted version that only makes Boolean decisions and inferences, and is designed for sparse parity equations. While our clause learning procedures are similar, we provide distinct contributions. On the theory side, we are able to prove that our algorithm *p*-simulates Res($\oplus$), using a new characterization of Res($\oplus$) that also illuminates how our clause learning algorithm is a direct analogue of classical 1-UIP learning. On the practical side, our implementation efficiently uses dense parity equations for both decisions and inferred constraints, allowing greater flexibility and potential to improve over classical CDCL on CNF formulas. We also propose new XNF-specific heuristics and incorporate proof logging.

Our contributions

Our algorithm CDCL($\oplus$) ("CDCL-parity") generalizes CDCL to XNF formulas, by branching on arbitrary parity constraints, performing linear-algebraic reasoning during unit propagation, and learning linear clauses through conflict analysis. The clause learning algorithm follows

naturally from a new characterization of the Res(⊕) proof system, which is our key technical tool. Also, although implication graphs are used heavily in classical CDCL, they have no clear XNF analogue. Instead, our definitions match what classical CDCL does semantically.

We prove that CDCL(⊕) *p*-simulates the Res(⊕) proof system using nondeterministic decisions and restarts. As a side note, this proof can also be adapted to classical CDCL and Resolution, and thereby improves the polynomial factor originally proven in [44], matching that in [7] with a simpler argument. We also prove that our unit propagation is equivalent to a fragment of Res(⊕) called input Res(⊕), again mimicking a result about classical CDCL [5].

We call our proof-of-concept implementation Xorcle (XOR Clause LEarning). In it, we adapt and introduce heuristics for watch structures, decision making, and phase selection, and additionally implement proof logging using a new LRUP(⊕) ("LRUP-parity") format. Experimental results show that despite being a new solver, Xorcle outperforms existing solvers such as Kissat, CryptoMiniSAT, and 2-Xornado on a selected suite of benchmarks, demonstrating the potential of our approach. The benchmark set consists of one cryptography-related XNF family [1], three synthetic XNF families, and one CNF family consisting of Tseitin formulas [53]. Tseitin formuals are notable because even without preprocessing, Xorcle's running time scales nearly polynomially on our test set, despite exponential lower bounds for Resolution and classical CDCL [54].

The full version of this paper [6] contains more proofs, details on heuristics and experiments, and semantic characterizations of classical CDCL that we mimic in CDCL(⊕).

2 Preliminaries on Res(⊕)

The set $\mathbb{F}_2 = \{0, 1\}$ is the field of two elements. We will write parity expressions as affine equations such as $[x \oplus y = 0]$ instead of $\neg(x \oplus y)$. The symbol $\oplus$ denotes XOR on individual Booleans, whereas $+$ denotes bitwise XOR on equations, where we associate an equation with a vector in $\mathbb{F}_2^{n+1}$ by reading off its coefficients, ending with the RHS. For example, with 2 variables, $[x = 1]$ is $(1, 0, 1)$, $[y = 1]$ is $(0, 1, 1)$, and $[x \oplus y = 0]$ is $(1, 1, 0)$. Then $[x = 1] + [y = 1] = [x \oplus y = 0]$. Additionally, we adopt the notation $\mathbf{1} = [0 = 1]$ (i.e. $\bot$ in Boolean contexts) and $\mathbf{0} = [0 = 0]$, so that if $f = [x \oplus y = 1]$, then $f + \mathbf{1} = [x \oplus y = 0]$.

When determining linear independence, we consider the full vector, so $\{[x = 0], [x = 1]\}$ is linearly independent. A set of equations is consistent if they have a common solution. The set of equations logically implied by a *consistent* starting set of equations U is exactly $\text{span}(U)$, the $\mathbb{F}_2$-linear span of U.

Next, we will define the Res(⊕) proof system [32, 33]. A proof is comprised of a finite sequence of lines, each of which is either a starting hypothesis or derived from previous lines via application of inference rules [16]. We call the lines of Res(⊕) *linear clauses*.

▶ **Definition 2.1.** *A* linear clause *is a disjunction of affine equations over* $\mathbb{F}_2$*. (We will often just call these clauses, unless we need to disambiguate them from Boolean clauses.) By convention, we disallow tautological clauses (evaluating to true on all assignments). We write* $C \equiv D$ *if the two clauses are true on the same assignments. A clause with one equation is called a* unit*, including the contradiction* $\mathbf{1}$*. An* XNF formula *is a conjunction of linear clauses, and a* k-XNF formula *is one where every linear clause has at most* k *equations.*

Linear clauses strictly generalize Boolean clauses, since one can take affine equations with a single variable. An important distinct feature is that two syntactically distinct linear clauses C and D may actually have $C \equiv D$. For example, $[x = 1] \vee [y = 1] \equiv [x \oplus y = 1] \vee [y = 1]$. In order to efficiently determine semantic equivalence, the procedure is to negate the clause

and take the span. For example, the negation of $C = [x = 1] \vee [y = 1]$ is $[x = 0] \wedge [y = 0]$. Because the linear span of a set of units has the same set of satisfying assignments as the set itself, $\text{span}([x = 0], [y = 0])$ captures the semantic information of C.

▶ **Definition 2.2.** *For a linear clause $C = f_1 \vee \cdots \vee f_m$, define its* linear negation *to be the subspace of affine equations* $\text{neg}(C) = \text{span}(f_1 + \mathbf{1}, \ldots, f_m + \mathbf{1})$.

▶ **Proposition 2.3.** *For any clauses C and D, we have $C \equiv D$ if and only if* $\text{neg}(C) = \text{neg}(D)$.

Note that the linear negation is the linear algebraic dual space to the set of *falsifying* assignments. Importantly, it is a subspace of equations, not assignments.

▶ **Definition 2.4** ([32])**.** *The* Res($\oplus$) *proof system has linear clauses as lines, and the following two inference rules:*

- *(resolution) From $C \vee f$ and $D \vee (f + \mathbf{1})$, deduce $C \vee D$.*
- *(weakening) From C, deduce any D such that $C \Rightarrow D$, equivalently* $\text{neg}(C) \subseteq \text{neg}(D)$.

For example, one can weaken $[x \oplus y = 1]$ to deduce $[x = 1] \vee [y = 1]$. The sheer number of possibilities allowed by weakening makes it difficult to create a solver that p-simulates Res($\oplus$). Although there are results that show how weakening can be limited [32, 31], our key insight is a new equivalent ruleset for Res($\oplus$) that has no weakening at all, which we state in the following theorem. We will prove it after a necessary definition.

▶ **Theorem 2.5.** *The following two inference rules are polynomially equivalent to resolution and weakening for the purpose of refutation (i.e., proving unsatisfiability).*

- *(addition) From $C \vee f$ and $D \vee g$, deduce $C \vee D \vee (f + g)$.*
- *(change of basis) From C, deduce any D such that $C \equiv D$, equivalently* $\text{neg}(C) = \text{neg}(D)$.

The simulation preserves graph structure, thus tree-like fragments also simulate each other.

Note that addition generalizes resolution by taking $g = f + \mathbf{1}$. One may still be concerned because the number of possible change of basis operations remains exponential in the width (number of equations) of the clause. However, our algorithms will be efficient because it suffices to use change of basis in a very specific way that is also used in proving Theorem 2.5.

▶ **Definition 2.6.** *Let $U \cup \{f\}$ be a set of equations and let C be a clause. We say that C* uses *f (with respect to U) if either* $\text{neg}(C) \subseteq \text{span}(U, f)$ *or* $\text{neg}(C) \subseteq \text{span}(U, f + \mathbf{1})$*, but* $\text{neg}(C) \not\subseteq \text{span}(U)$*. If C uses f,* isolating *f refers to changing the basis of C to $C' \vee g$, where $C \equiv C' \vee g$ and* $\text{neg}(C') \subseteq \text{span}(U)$*; we say that f is* isolated (with respect to U) *afterwards.*

For example, take $U = \{[x = 0], [y = 0]\}$, $f = [a = 0]$, and $C = [x \oplus a = 1] \vee [y \oplus a = 1]$. One can check that C uses f w.r.t. U, but f is not isolated because it appears in both equations of C. To isolate f, because $\text{span}([x \oplus a = 0], [y \oplus a = 0]) = \text{span}([x \oplus y = 0], [y \oplus a = 0])$, the clause $[x \oplus y = 1] \vee [y \oplus a = 1]$ isolates f with respect to U.

More generally, if C uses f with respect to U, let $\text{span}(f_1, \ldots, f_k, f_{k+1}, \ldots, f_m)$ be any basis of $\text{neg}(C)$, where the f_i are ordered so that $f_i \in \text{span}(U)$ iff $i \leq k$. (In other words, $f_{k+1}, \ldots, f_m$ involve the extra equation f.) Then the following isolates f.

$$\underbrace{(f_1 + \mathbf{1}) \vee \cdots \vee (f_k + \mathbf{1}) \vee (f_{k+1} + f_m + \mathbf{1}) \vee \cdots \vee (f_{m-1} + f_m + \mathbf{1})}_{C'} \vee \underbrace{(f_m + \mathbf{1})}_{g}$$

Proof of Theorem 2.5. First, change of basis is a restricted form of weakening, and addition is simulated by weakening $C \vee f$ into $C \vee (f+g) \vee (g+\mathbf{1})$, then resolving with $D \vee g$ to get $C \vee D \vee (f+g)$. This proves the soundness of these inference rules, as well as one direction of the simulation.

In the other direction, we will convert any proof using resolution/weakening to a proof using addition/change of basis. Proceeding in topological order, we will maintain the structure and convert individual lines, where each new line will be at least as strong as the corresponding old one, which is sufficient for refutation. The base cases are the starting clauses, which remain the same.

In the inductive step, consider the first not-yet-converted clause derived via resolution: $\hat{C} \vee \hat{D}$ derived from $\hat{C} \vee (f+\mathbf{1})$ and $\hat{D} \vee f$. Then, clauses $\hat{C} \vee (f+\mathbf{1})$ and $\hat{D} \vee f$ must be derived by sequences of weakenings from already-converted clauses C and D, respectively, using the fact that each original clause can be seen as a weakening of its corresponding converted clause.

Let U be a basis of $\mathrm{neg}(\hat{C} \vee \hat{D}) = \mathrm{span}(\mathrm{neg}(\hat{C}), \mathrm{neg}(\hat{D}))$. Since $\hat{C} \vee (f+\mathbf{1})$ can be derived from C by a sequence of weakenings, we have $\mathrm{neg}(C) \subseteq \mathrm{span}(\mathrm{neg}(\hat{C}), f) \subseteq \mathrm{span}(U, f)$. We may also assume that $\mathrm{neg}(C) \not\subseteq \mathrm{span}(U)$; otherwise, take C itself as the conversion of $\hat{C} \vee \hat{D}$. Thus, C uses f with respect to U. One similarly sees that D uses f.

This setup lets us isolate f in both C and D, creating clauses $C' \vee g$ and $D' \vee h$ such that $C \equiv C' \vee g$, $D \equiv D' \vee h$, and $\mathrm{neg}(C'), \mathrm{neg}(D') \subseteq \mathrm{span}(U)$. (This is change of basis.) Then, we apply the addition rule on $C' \vee g$ and $D' \vee h$, resulting in $C' \vee D' \vee (g+h)$.

By construction, the coefficient of f is 1 when $g+\mathbf{1}$ is written in the $U \cup \{f\}$ basis, and similarly the coefficient of $f+\mathbf{1}$ is 1 when $h+\mathbf{1}$ is written in the $U \cup \{f+\mathbf{1}\}$ basis. Thus, $g+h+\mathbf{1} \in U$, and $\mathrm{neg}(C' \vee D' \vee (g+h)) \subseteq \mathrm{span}(U)$ which equals $\mathrm{neg}(\hat{C} \vee \hat{D})$. Therefore $C' \vee D' \vee (g+h)$ is at least as strong as $\hat{C} \vee \hat{D}$ as required. ◀

The key steps in the proof above are shown in the example below. Intuitively, the equation that we isolate and cancel out via addition is analogous to a variable being resolved on.

▶ **Example 2.7.** Consider the Res($\oplus$) proof below whose resolution step is highlighted.

$$\dfrac{\dfrac{[x \oplus a = 1] \vee [y \oplus a = 1]}{[x=1] \vee [y=1] \vee \boxed{[a=1]}}\,\text{weak} \qquad \dfrac{[z \oplus a = 0] \vee [w \oplus a = 0]}{[z=1] \vee [w=1] \vee \boxed{[a=0]}}\,\text{weak}}{[x=1] \vee [y=1] \vee [z=1] \vee [w=1]}\,\text{res}$$

Weakening allowed the proof to resolve $[a=0]$ and $[a=1]$. Instead, in the revised proof below, we will isolate a on both sides, allowing it to be canceled out by addition.

$$\dfrac{\dfrac{[x \oplus a = 1] \vee [y \oplus a = 1]}{[x \oplus y = 1] \vee \boxed{[y \oplus a = 1]}}\,\text{change-basis} \qquad \dfrac{[z \oplus a = 0] \vee [w \oplus a = 0]}{[z \oplus w = 1] \vee \boxed{[w \oplus a = 0]}}\,\text{change-basis}}{[x \oplus y = 1] \vee [z \oplus w = 1] \vee \boxed{[y \oplus w = 1]}}\,\text{add}$$

Though this final result is not exactly identical to the result from the original Res($\oplus$) proof, it implies the original conclusion and is therefore better.

Finally, we make a note that one can formalize the idea that change of basis is only used to isolate, and that addition is only used to cancel out. We call this *affine resolution*.

Algorithm 1 Basic CDCL(⊕).

Input: A set of linear clauses Δ

```
Initialize U ← ∅, R ← ∅.      // True units and corresponding reason clauses
while true do
    Run unit propagation on Δ (Section 3.1) and update U and R accordingly.
    if 1 ∈ span(U) then
        if U contains a decision then
            Find an asserting clause C (Section 3.2) and add it to Δ.
            Revert U and R to the asserting level of C.
        else
            return "unsatisfiable"
    else
        if |U| < n then
            Restart or decide any f with f, f + 1 ∉ span(U) (Section 5) to add to U.
        else
            return "satisfiable" and the solution to the system of equations U
```

▶ **Definition 2.8.** *The* affine resolution *rule may be applied to clauses C and D whenever there exists $U \cup \{f\}$ such that both C and D use f, and specifically* $\mathrm{neg}(C) \subseteq \mathrm{span}(U, f)$ *while* $\mathrm{neg}(D) \subseteq \mathrm{span}(U, f + \mathbf{1})$ *or vice versa. The rule derives any clause with linear negation* $\mathrm{span}(\mathrm{neg}(C), \mathrm{neg}(D), \mathbf{1}) \cap \mathrm{span}(U)$.

▶ Remark 2.9. The affine resolution rule by itself is polynomially equivalent to Res(⊕).

Although the definition is abstract, it is identical to the process we have been describing.

▶ **Proposition 2.10.** *The result of affine resolution may be obtained by isolating f to get $C \equiv C' \vee g$ and $D \equiv D' \vee h$, where* $\mathrm{neg}(C'), \mathrm{neg}(D') \subseteq \mathrm{span}(U)$*, then taking $C' \vee D' \vee (g + h)$. In other words,* $\mathrm{neg}(C' \vee D' \vee (g + h)) = \mathrm{span}(\mathrm{neg}(C), \mathrm{neg}(D), \mathbf{1}) \cap \mathrm{span}(U)$.

Proof sketch. The $\subseteq$ direction is quickly checked by definitions. For the $\supseteq$ direction, consider any equation f^* in the RHS and note that we have $\mathrm{span}(\mathrm{neg}(C), \mathrm{neg}(D), \mathbf{1}) = \mathrm{span}(\mathrm{neg}(C'), g + \mathbf{1}, \mathrm{neg}(D'), h + \mathbf{1}, \mathbf{1})$. Write f^* in this decomposition; then, because f^* must reduce to 0 modulo $\mathrm{span}(U)$, one can calculate that the coefficients of $g + \mathbf{1}$, $h + \mathbf{1}$, and $\mathbf{1}$ must be equal. This implies that f^* belongs to the LHS. ◀

3 The CDCL(⊕) Algorithm

In this section, we define an algorithm, CDCL(⊕), that mimics classical CDCL in the Res(⊕) proof system. Its basic form is given in Algorithm 1, calling subroutines for unit propagation and clause learning, as well as decision heuristics. The correctness of this algorithm is argued identically to the correctness of classical CDCL. Note that we include optional restarts in this basic form (i.e. deleting all decisions made so far) because they will be necessary for Theorem 4.10, on the simulation of Res(⊕) by CDCL(⊕). In the rest of this section, we will explain the unit propagation and clause learning subroutines.

3.1 Unit propagation

Let U be a list of equations (i.e. units) and Δ be a set of clauses. We will ensure that U is always linearly independent (analogous to not repeating literals), and is consistent until the conflict is deduced (analogous to never having both a literal and its negation).

Algorithm 2 Unit propagation on a particular clause.

Input: Set of starting units U and clause C

Write $\text{neg}(C) = \text{span}(f_1, \ldots, f_m)$.
Write U in row-echelon form (i.e. each equation as a row, with unique leading terms).
Define $g_i \leftarrow f_i$ reduced by U for each i (i.e. eliminate all variables in f_i that are leading variables in U, by adding equations from U).
if *some* g_i *is* $\mathbf{1}$ **then**
  return *no propagation*
else if *every* g_i *is* $\mathbf{0}$ **then**
  return $\mathbf{1}$
else if *there exists* k *such that every* g_i *is either* $\mathbf{0}$ *or identically* g_k **then**
  return $f_k + \mathbf{1}$
else
  return *no propagation*

Because we want CDCL($\oplus$) to be *semantically* similar to CDCL, we want unit propagation to deduce all units that are *logically implied* by the combination of a clause and some known units. This will allow us to derive more units than if we tried to be *operationally* similar; i.e., checking if all but one equation in a clause consists of negations of known units.

▶ **Example 3.1.** Suppose that $U = \{[x = 0], [y = 0]\}$, and we are processing the clause $C = [x \oplus y \oplus z = 1] \vee [z = 1]$. By plugging the known information into the clause, we get $[z = 1] \vee [z = 1]$, so the unit $[z = 1]$ is true. This shows that if we want to determine if a clause reduces to a unit, it is insufficient to simply check for negations of known units, and some linear algebraic reasoning is necessary.

Also, because we deduce $[z = 1]$ and already know U, the result is that we know all equations in $\text{span}(U, [z = 1]) = \text{span}([x = 0], [y = 0], [z = 1])$ to be true, such as $[x \oplus y \oplus z = 1]$. Then, instead of choosing to deduce $[z = 1]$, one can see that it would have been equivalent to deduce from C any equation in $\text{span}(U, [z = 1]) \setminus \text{span}(U)$, since it would produce the same span when added to U.

The definition below incorporates the considerations in Example 3.1 at a high level, and coincides with the definition of unit propagation for the SRES proof system from [30].

▶ **Definition 3.2.** *From a set of starting units* U*,* unit propagation on a clause C *refers to:*

1. *Deducing the contradiction* $\mathbf{1}$ *if* $\text{neg}(C) \subseteq \text{span}(U)$.
2. *Otherwise, deducing an equation* f *if* $\text{neg}(C) \subseteq \text{span}(U, f + \mathbf{1})$ *and* f, $f + \mathbf{1} \notin \text{span}(U)$.

We call C *the* reason clause *for* f*, if* f *is deduced (or if* $\mathbf{1}$ *is deduced) as above.*

Note that the definition relies only $\text{span}(U)$*, not the exact units in* U*. Thus, we may refer to a starting subspace (rather than a starting set of units), since every basis would produce the same propagation.*

As written, the definition checks exponentially many equations f and is also non-deterministic, allowing many different f to be deduced. This is not a problem, as we provide a polynomial-time, canonically deterministic algorithm in Algorithm 2.

▶ **Proposition 3.3.** *The following properties of unit propagation are true:*

1. *If unit propagation on a clause* $C = f_1 + \mathbf{1} \vee \cdots \vee f_m + \mathbf{1}$ *permits deducing an equation, then it permits deducing some* $f_i + \mathbf{1}$ *in particular (for poly-time deduction).*

2. *If f and g can be deduced, then* $\text{span}(U, f) = \text{span}(U, g)$ *(for canonical determinism). These show that Algorithm 2 implements Definition 3.2 up to equivalent span.*

Proof sketch. For (1), if we can deduce f, write $f_1, \ldots, f_m$ in the $U \cup \{f + \mathbf{1}\}$ basis. Then one can check that any $f_i + \mathbf{1}$ where f_i has non-zero $f + \mathbf{1}$ component can be deduced.

For (2), we know $f, g \in \text{neg}(C) \setminus \text{span}(U)$, and are thus equal mod $\text{span}(U)$ (i.e. in the quotient space) because the subspace $\text{neg}(C)$ only extends beyond $\text{span}(U)$ in one dimension.

To conclude that Algorithm 2 is correct, note that it checks if $\text{neg}(C) \equiv \text{span}(f_i) \pmod{\text{span}(U)}$ for each i, and this holds iff $\text{neg}(C) \subseteq \text{span}(U, f_i)$ as desired. ◀

Note that our actual implementation of unit propagation includes a few optimizations over Algorithm 2. For one, we keep U in row-echelon form between runs instead of recomputing it each time. This is also why the formal definition of unit propagation is nondeterministic, allowing us to choose equivalent equations to learn as long as we maintain the same span. Finally, the definition considers arbitrary f instead of only equations appearing in the clause C, since this property is important for establishing the semantic characterization below.

▶ **Proposition 3.4.** *Let U be a consistent set of units. Unit propagation on a clause C*

- *deduces the contradiction $\mathbf{1}$ if and only if $C \wedge U \Rightarrow \mathbf{1}$, and*
- *deduces an equation f if and only if $C \wedge U \Rightarrow f$, $C \wedge U \not\Rightarrow \mathbf{1}$, and $f, f + \mathbf{1} \notin \text{span}(U)$.*

Finally, unit propagation on a *set* of clauses is defined analogously to the classical case.

▶ **Definition 3.5.** Unit propagation *on a set of clauses Δ refers to choosing clauses $C \in \Delta$ and updating U with unit propagation on C, until $\mathbf{1} \in U$ or no $C \in \Delta$ causes propagation.*

We note that U stays linearly independent and consistent while deducing equations $f \neq \mathbf{1}$ because we require $f, f + \mathbf{1} \notin \text{span}(U)$. When $\mathbf{1}$ is explicitly deduced and added, the set U stays linearly independent but becomes inconsistent. Lastly, the important property of a unique fixed point is preserved in our setting, allowing clauses C to be chosen in arbitrary order, proven similarly to the classical case.

▶ **Proposition 3.6.** *Let Δ be a set of clauses and let U be a set of equations. Let U_* and U'_* be two potential results from unit propagation on Δ starting with U. Then either* $\mathbf{1} \in \text{span}(U_*) \cap \text{span}(U'_*)$ *or* $\text{span}(U_*) = \text{span}(U'_*)$.

3.2 Clause learning

In classical CDCL, clause learning is often defined using implication graphs, which draw directed edges (x, y) between units if x is part of the reason for deducing y in unit propagation. However, the graph structure does not translate easily to CDCL($\oplus$). For example, continuing Example 3.1, it is not clear which edges should point into $[z = 1]$. If one creates edges from both $[x = 0]$ and $[y = 0]$, attempting to capture that these two units were used to deduce $[z = 1]$, the graph only encodes the information $[x = 1] \vee [y = 1] \vee [z = 1]$, which is weaker than the original clause $C = [x \oplus y \oplus z = 1] \vee [z = 1]$ used to deduce $[z = 1]$.

Instead, we use a semantically motivated definition. To set up the notation, let $U = f_1, \ldots, f_m, f_{m+1}$ with $f_{m+1} = \mathbf{1}$ be a sequence of unit propagations and decisions ending in contradiction, and let $R = R_1, \ldots, R_m, R_{m+1}$ with $R_{m+1} = R_\perp$ be the corresponding list of reason clauses, including decisions denoted $R_i = *$. Write $U_k = f_1, \ldots, f_k$ for the first k units and define $\text{level}(f_i)$ as the number of decisions in $R_1, \ldots, R_i$.

Algorithm 3 1-UIP–like clause learning.

Input: Units $U = f_1, \ldots, f_m, \mathbf{1}$ and corresponding reasons $R = R_1, \ldots, R_m, R_\perp$

Initialize $C \leftarrow R_\perp$ and $i \leftarrow m$.
// Invariant: $\mathrm{neg}(C) \subseteq \mathrm{span}(U_i)$ and R_i is not a decision.
while C *is not asserting* **do**
  Decrement i until C uses f_i (i.e. $\mathrm{neg}(C) \subseteq \mathrm{span}(U_i)$ but $\mathrm{neg}(C) \not\subseteq \mathrm{span}(U_{i-1})$).
  Isolate f_i in C by changing basis into $C' \vee f$ with $\mathrm{neg}(C') \subseteq \mathrm{span}(U_{i-1})$.
  // f contains f_i, possible by line 3
  Isolate f_i in R_i by changing basis into $R_i' \vee g$ with $\mathrm{neg}(R_i') \subseteq \mathrm{span}(U_{i-1})$.
  // g contains f_i, possible by definition of unit propagation
  Apply addition to update $C \leftarrow C' \vee R_i' \vee (f + g)$.
  // Now $\mathrm{neg}(C) \subseteq \mathrm{span}(U_{i-1})$ because f_i's in f and g cancel out
return C

▶ **Definition 3.7.** *A clause C is a* conflict clause *(with respect to U and R as above) if*
- $\mathrm{neg}(C) \subseteq \mathrm{span}(U_m)$ *and*
- *unit propagation starting with units* $\mathrm{neg}(C)$ *and clauses R produces a contradiction.*

The asserting level *of a conflict clause C is the smallest level k such that starting with the prefix of U consisting all units of level at most k, unit propagation on C results in some propagation. A conflict clause C is* asserting *if its asserting level is less than* $\mathrm{level}(\mathbf{1})$.

In classical CDCL, a 1-UIP asserting clause is found by taking the reason for contradiction $R_\perp$ and going backwards, successively resolving with other reason clauses on the propagated variables. In CDCL($\oplus$), Algorithm 3 mimics the same algorithm using affine resolution instead, though we state it in terms of addition and change of basis to be more concrete.

▶ **Proposition 3.8.** *Algorithm 3 computes an asserting clause.*

Proof sketch. The invariant $\mathrm{neg}(C) \subseteq \mathrm{span}(U_i)$ is justified in the pseudocode comments, and unit propagation from $\mathrm{neg}(C)$ and R derives contradiction by replaying the original propagation, so C is a conflict clause. If k is the index of the last decision and the loop reaches k, then C must be asserting because $\mathrm{neg}(C) \subseteq \mathrm{span}(U_k) = \mathrm{span}(U_{k-1}, f_k)$ where U_{k-1} is the previous level. So C can be used to propagate, in particular deducing $f_k + \mathbf{1}$. ◀

Note also that although there are generally many ways to isolate an equation in a clause, the determinism of Algorithm 3 is given by Proposition 2.10. Below, we give an example of Algorithm 3 in action.

▶ **Example 3.9.** Suppose that we decide equations f_1 and f_2, then unit propagation produces f_3, f_4, and $\mathbf{1}$ using reason clauses $R_3 = (f_2 + f_3) \vee (f_1 + f_3)$, $R_4 = (f_1 + f_3 + f_4) \vee (f_3 + \mathbf{1})$, and $R_\perp = (f_1 + f_4 + \mathbf{1}) \vee (f_3 + f_4 + \mathbf{1})$. Note that we have already converted all reason clauses into the U basis, which is necessary to efficiently isolate f_i's during the algorithm.

To find an asserting clause, start with $R_\perp$. This clause is not asserting because no unit propagation is possible from just f_1. In the main loop, we generate the following Res($\oplus$) proof, which isolates and eliminates f_4. Recall that change of basis for $R_\perp$ operates on $\mathrm{neg}(R_\perp)$, where $\mathrm{span}(f_1 + f_4, f_3 + f_4) = \mathrm{span}(f_1 + f_3, f_3 + f_4)$. For the other side, observe that R_4 already has f_4 isolated, so no change of basis is needed.

$$\dfrac{\dfrac{(f_1 + f_4 + \mathbf{1}) \vee (f_3 + f_4 + \mathbf{1})}{(f_1 + f_3 + \mathbf{1}) \vee (f_3 + f_4 + \mathbf{1})}\,\text{change-basis} \qquad (f_1 + f_3 + f_4) \vee (f_3 + \mathbf{1})}{(f_1 + f_3 + \mathbf{1}) \vee (f_3 + \mathbf{1}) \vee (f_1 + \mathbf{1})}\,\text{add}$$

The final clause is linearly dependent and simplifies to $(f_1 + \mathbf{1}) \vee (f_3 + \mathbf{1})$, which is asserting because $f_3 + \mathbf{1}$ can be deduced by unit propagation from f_1. Note that R_3 was not used at all, indicating that Algorithm 3 can terminate before reaching the decision.

4 Connections to Res($\oplus$)

To recap, there are two important ways that classical CDCL relates to Resolution. First, CDCL with any asserting clause learning procedure (not just the 1-UIP algorithm) produces Resolution proofs of size at most the number of unit propagations [5]. More generally, this can be viewed as an exact equivalence between unit propagation and a fragment of Resolution known as input Resolution (or sometimes trivial Resolution) [12, 5]. Second is the converse direction, that any problem with a Resolution proof of unsatisfiability of length M also admits a sequence of decisions and restarts for CDCL that returns unsatisfiable after learning at most $O(n^2 M)$ clauses [44, 7]. We generalize both of these to CDCL($\oplus$) and Res($\oplus$).

For simplicity in this section, consider the "lines" of Res($\oplus$) proofs to be consistent subspaces of equations (i.e. $\text{neg}(C)$ instead of C, so that a *consistent* subspace corresponds to a *non-tautological* clause). Then change of basis is the identity operation, so the only inference rule is addition: from $\text{span}(A, f)$ and $\text{span}(B, g)$, deduce $\text{span}(A, B, f + g + \mathbf{1})$.

▶ **Definition 4.1.** *An* input Res($\oplus$) proof *of a clause C from a set of clauses Δ is a sequence of subspaces $\text{neg}(C_0), \text{neg}(C_1), V_1, \text{neg}(C_2), V_2, \text{neg}(C_3), V_3, \ldots, V_k$ with $V_k = \text{neg}(C)$ that is itself a valid* Res($\oplus$) *proof, where each $C_i \in \Delta$ and each V_i is derived from an appropriate addition of V_{i-1} and $\text{neg}(C_i)$, considering $V_0 = \text{neg}(C_0)$. The length of the proof is k.*

▶ **Definition 4.2.** *A consistent subspace of equations V is* conflict-like with respect to a set of clauses Δ *if unit propagation starting with units V and clauses Δ produces a contradiction.*

▶ **Theorem 4.3.** *If a clause C has an input* Res($\oplus$) *proof from Δ, then $\text{neg}(C)$ is conflict-like with respect to Δ. Conversely, if $\text{neg}(C)$ is conflict-like, there is an input* Res($\oplus$) *proof of C' such that $\text{neg}(C') \subseteq \text{neg}(C)$. The number of additions in the proof is equivalent to the number of unit propagations made.*

While the version of Theorem 4.3 for input Resolution and classical CDCL was originally proved using implication graphs [5], one can alternatively prove it by constructing a run of unit propagation for one direction and running Algorithm 3 without stopping for asserting clauses in the other direction, which generalizes to our setting. Also, because every conflict clause is also conflict-like, we get the following corollary, establishing one direction of the relationship between CDCL($\oplus$) and Res($\oplus$).

▶ **Corollary 4.4.** *Let Δ be an unsatisfiable set of clauses and consider* CDCL($\oplus$) *with any asserting clause learning scheme. The execution of the algorithm on Δ may be converted into a* Res($\oplus$) *proof of unsatisfiability with at most as many additions as the number of unit propagations made.*

For the converse relationship, the original simulation of Resolution by CDCL in [44] used a concept called *absorption* [43]. A Boolean clause C is *absorbed* by a set of clauses Δ if for all sets of starting units U, unit propagation on the set $\Delta \cup \{C\}$ deduces the same units

as Δ alone. (In some treatments, a clause that is not absorbed is called "1-empowered," though we won't use this phrase.) This definition translates easily to CDCL($\oplus$) and its more powerful unit propagation algorithm, but it turns out to be the wrong definition to use.

▶ **Example 4.5.** Suppose that $\Delta = \{[x = 1] \vee [z = 0], [y = 1] \vee [z = 1]\}$ and we are considering $C = [x = 1] \vee [y = 1]$. This same example is a valid input for both classical unit propagation and the unit propagation in CDCL($\oplus$). Classically, C would be absorbed by Δ because the only ways to use C are if $[x = 0]$ or $[y = 0]$ were part of the starting units U, and in both of these cases, the deduction using C can also be deduced using the two clauses in Δ.

However, in our version, suppose that $U = \{[x \oplus y = 0]\}$. We see that C is actually not absorbed, because unit propagation can deduce, for example, $[y = 1]$ from U and C, whereas nothing can be unit propagated from U and Δ. In other words, it is much harder to absorb clauses in our version, because unit propagation is more powerful. This prevents us from tracing the argument in [44] directly.

Instead of absorbing clauses, we absorb specific decompositions of clauses. Intuitively, a decomposition is a particular way that a clause can be used for unit propagation, and it is absorbed if that method can be done through Δ instead. Absorbing a clause is equivalent to absorbing every possible decomposition. While there are at most n possible decompositions for Boolean clauses, there are exponentially many possible decompositions for linear clauses.

▶ **Definition 4.6.** *Let V be a consistent subspace of equations and let Δ be a set of of clauses. Let (V', f) be such that $V = \text{span}(V', f)$. We call this a* decomposition *of V. The decomposition is* absorbed *by Δ if unit propagation with units V' and clauses Δ either results in contradiction or deriving $f + \mathbf{1}$.*

The addition rule of Res($\oplus$) naturally specifies decompositions of its antecedents, and these decompositions used in a given Res($\oplus$) proof are exactly what we absorb. Then, at a high level, the rest of the construction of decisions and restarts to absorb these decompositions closely follows closely the original argument in [44], and in fact improves it by a multiplicative factor n (the number of variables).

To summarize the improvement, given a clause C in a Resolution proof of unsatisfiability from Δ, the authors in [44] force the solver to absorb an entire conflict-like clause C, equivalent to absorbing all n decompositions in the Boolean case. Instead, we only absorb the decompositions that are used in applications of the addition rule, which on average is fewer than 2 decompositions per clause, a factor n improvement. For completeness, we give the full argument below, following the same definition and lemma structure of [44], which starts by defining which decompositions we would like to absorb.

▶ **Definition 4.7.** *Let Δ be a set of clauses and V be a conflict-like space of equations. A decomposition (V', f) of V is* asserting-like for Δ *if (V', f) is not absorbed by Δ.*

▶ **Proposition 4.8.** *Suppose that Δ is unsatisfiable and unit propagation alone cannot derive a contradiction. Then every* Res($\oplus$) *refutation of Δ contains a deduction*

$$\frac{\text{span}(A, f) \qquad \text{span}(B, g)}{\text{span}(A, B, f + g + \mathbf{1})}$$

such that at least one of (A, f) and (B, g) is asserting-like for Δ.

Proof. Our refutation starts with linear negations of clauses and ends with $\mathbf{0} = [0 = 0]$. Note that starting subspaces are conflict-like, and by hypothesis, $\mathbf{0}$ is not conflict-like. Consider the deduction of the first subspace in the proof that is not conflict-like, with notation as

in the proposition statement. Then, (A, f) and (B, g) are conflict-like, and suppose for contradiction that both are absorbed. This means that A and B either deduce contradiction, or deduce $f + \mathbf{1}$ and $g + \mathbf{1}$, respectively. All cases contradict $\text{span}(A, B, f + g + \mathbf{1})$ not being conflict-like. ◀

Note that once every asserting decomposition in a proof is absorbed by Δ, the contrapositive of Proposition 4.8 implies that unit propagation will deduce contradiction from Δ. Thus, we turn our attention to absorbing asserting decompositions. We adopt the convention that when we specify a sequence of decisions, some decisions may already be contained in $\text{span}(U)$, and in this case CDCL($\oplus$) automatically skips them.

▶ **Proposition 4.9.** *Let Δ be a set of clauses and let (V', f) be an asserting-like decomposition for a consistent subspace of equations V. There exists a decision sequence that results in absorbing (V', f) while learning at most n^2 clauses, where n is the number of variables.*

Proof. Until (V', f) is absorbed, we repeatedly perform the following loop as our decision sequence for CDCL($\oplus$): decide a restart, then any basis of V', then f. We claim that we learn n clauses per loop iteration, and the loop terminates within n iterations.

For the number of clauses per loop iteration, first note that we never deduce contradiction and perform conflict analysis before deciding f because (V', f) is not absorbed, and we always deduce contradiction after deciding f because V is conflict-like. After reaching a contradiction, CDCL($\oplus$) can reach contradiction and learn clauses many more times before needing the next decision. Each contradiction moves the solver up by at least one level, so it takes at most n learned clauses to reach the next decision.

For the number of loop iterations, consider the first asserting clause derived in each iteration. Each allows a new equation to be deduced from decisions V'. Since each is new with respect to all previous learned asserting clauses, they must be linearly independent, and hence this can occur at most n times. ◀

▶ **Theorem 4.10.** *Given a* Res($\oplus$) *proof of unsatisfiability of a set of clauses Δ using M additions, there is a sequence of decisions and restarts such that* CDCL($\oplus$) *learns at most $2n^2M$ clauses before concluding unsatisfiability.*

Proof. The decision sequence is constructed adaptively while running CDCL($\oplus$). Until unit propagation can immediately derive contradiction from Δ, we repeatedly perform the following loop: use Proposition 4.8 to find an asserting-like decomposition (V', f), then use Proposition 4.9 to absorb (V', f) by learning n^2 new clauses.

Note that Δ is growing throughout this process, but Proposition 4.8 may view the proof of unsatisfiability as always starting with the original Δ and ignoring learned clauses. Since each absorbed decomposition must stay absorbed as Δ grows, and each addition uses 2 decompositions, the procedure must terminate within $2M$ iterations. ◀

Since each clause has a length $O(n)$ input Res($\oplus$) proof by Theorem 4.3, the translation of a Res($\oplus$) proof to a CDCL($\oplus$) execution and back to Res($\oplus$) incurs a multiplicative $O(n^3)$ overhead. When this argument can be translated line-by-line back to the Resolution and classical CDCL case, this $O(n^3)$ factor matches what is proven in [7].

▶ **Corollary 4.11.** *Theorem 4.10 is also true for Resolution and classical CDCL.*

5 Implementation and Heuristics

Our proof-of-concept implementation of CDCL(⊕) is called Xorcle. In this section, we summarize techniques and heuristics that Xorcle implements to compute efficiently, and note a few cases of common classical CDCL heuristics that are difficult to translate. This summary focuses on techniques and heuristics that are enabled by default in Xorcle. We selected these defaults based on experimentation over the families of XNF and CNF formulas described in Section 6.

Fast linear algebra over $\mathbb{F}_2$. We store equations with n variables as $\lceil (n+1)/64 \rceil$-length row vectors of bit-packed unsigned integers, using the `bit` library [22] with additional customizations. This representation makes it easy to add equations or reduce modulo a subspace in row-echelon form. The use of bit-packing for XOR constraints is similar to [47].

For clause learning as in Algorithm 3, note that when U is kept in row-echelon form (not reduced row-echelon form), we can preserve the sequence of subspaces $U_1, \ldots, U_m$ by tracking the order in which rows are inserted into U. This allows us to change the basis of every clause into the $U_{\text{reverse}} = \mathbf{1}, f_m, \ldots, f_1$ basis. Row-echelon form in the new basis guarantees that the latest propagated equation is already isolated.

Compaction. When the trail at decision level 0 is non-empty, we remove all satisfied clauses and reduce equations in all remaining clauses using this knowledge, removing the columns that are now zero. We call this *compaction*.

While generally a useful optimization, we note that proofs of unsatisfiability are much longer when compaction is on. This is because when compaction is on, learned clauses may be "missing" level 0 units, which need to be added back for proof logging. See Section 7 for details on proof logging.

Watched equations and clause selection. Watched literals [40] optimize classical unit propagation in two ways: (1) if neither of the two "watched" literals in a clause is falsified then it doesn't propagate, and (2) the "watchlists" of clauses that watch a particular literal allow the solver to quickly find candidate clauses for propagation.

CDCL(⊕) can easily watch two equations $f, g \in \text{neg}(C)$, with a note that we must also check if $f + g \in \text{span}(U)$ to determine if a watch must be updated. However, watchlists are more difficult to adapt. A new unit alone does not determine which clauses require action – it also depends on the previously known units, since linear combinations are allowed. Thus, we need heuristics for clause selection.

In experimental testing, it was fastest to simply cycle through all clauses in the order they appear in the input or are learned. While we attempted two alternative watchlist-inspired designs that sought to prefer or guarantee that selected clauses will be useful to look at (i.e. produce unit propagation or a change in watched equation), the overhead of maintaining these structures exceeded the performance benefits.

Decision procedure. Unless CDCL(⊕) branches on equations, it cannot improve over classical CDCL for CNF inputs, so we need to choose equations rather than variables. Classical SAT solvers commonly choose variables based on information tracked for each one, using techniques like VSIDS [40, 21] and CHB [37]; since we have exponentially many equations, these are difficult to adapt.

Instead, we choose an equation to branch on by first choosing a clause C and then choosing an equation associated with C, similar to some older approaches in classical SAT solving such as BerkMin [25] and Clause-Move-To-Front (CMTF) [23]. We cannot simply choose one of the equations syntactically represented in C, since that would still only branch on literals for CNF inputs. Thus our default approach is to branch on a random unfalsified equation in $\text{neg}(C)$ obtained by rejection sampling, justified by the following.

▶ **Proposition 5.1.** *Let U be a set of units and let C be a clause not satisfied by U such that there exists $g, h \in \text{neg}(C)$ with $g, h, g + h \notin \text{span}(U)$ (i.e. watched equations). For a uniformly random equation f sampled from the space $\text{neg}(C)$, there is a $\geq 75\%$ chance that f, $f + \mathbf{1} \notin \text{span}(U)$.*

It remains to choose the clause C. We introduce a new heuristic that we call CSIDS (Clause State Independent Decaying Sum), which is inspired by standard VSIDS and interpolates between the intuition behind clause choices in BerkMin and CMTF, generalizing both.

In particular, CSIDS maintains activity scores for all clauses and selects clauses in order of activity. Starting clauses have base activity 0, new asserting clauses have base activity $A_0 \geq 0$, each clause used during conflict analysis has activity bumped by $\Delta \geq 0$, and all activities decay by a multiplicative factor $0 \leq \alpha \leq 1$ at every conflict. Our default parameter choices are $A_0 = 1$, $\Delta = 1$, and $\alpha = 0.98$. Experimentally, this outperformed the BerkMin-like choice of $A_0 = 1$, $\Delta = 0$, and any $0 < \alpha < 1$, as well as the CMTF-like choice of $A_0 = 1$, $\Delta = 1$, and any $0 < \alpha < 0.5$. It also outperformed a baseline approach that mimics standard variable-based VSIDS and makes only variable decisions.

Phase selection. Common techniques such as phase saving [42] are also difficult to generalize due to tracking information for each variable. After sampling an equation $f \in \text{neg}(C)$ via the above, we tested the three simple techniques: always satisfy the clause (choosing $f + \mathbf{1}$), always falsify the clause (choosing f and making progress towards unit propagation), and randomly choosing. Experimentally, always falsifying performed best.

6 Experimental Results

In this section, we compare the performance of Xorcle with various heuristics and against other existing solvers. All experiments are run on an Apple M2 processor with 16 GB RAM and a 60 second timeout. Our benchmark families are listed below. Note that because XNF solving is still relatively underdeveloped, we were only able to find one industrial XNF family. Pebbling and Tseitin instances described below are generated using `cnfgen` [36].

Ascon-128 [1] Set of 400 satisfiable instances relating to the Ascon-128 cipher. Note that these are all 2-XNFs, specifically designed for the 2-XNF solver 2-Xornado.

Random k-XNFs The number of clauses is set so that about 50% of instances are satisfiable.

Restricted random k-XNFs Inspired by a technique in approximate model counting [11], we restrict a random k-XNF by a system of random linear equations. For Xorcle, the linear equations should not increase the difficulty of solving.

Pebbling formulas lifted by k-XORs Pebbling formulas are unsatisfiable and can be solved using unit propagation alone. We replace each variable with the XOR of k fresh variables. Xorcle continues to solve the lifted version using just unit propagation.

Tseitin formulas in CNF An unsatisfiable family consisting of parity expressions naively translated to CNF. They are provably difficult for Resolution-based solvers [54].

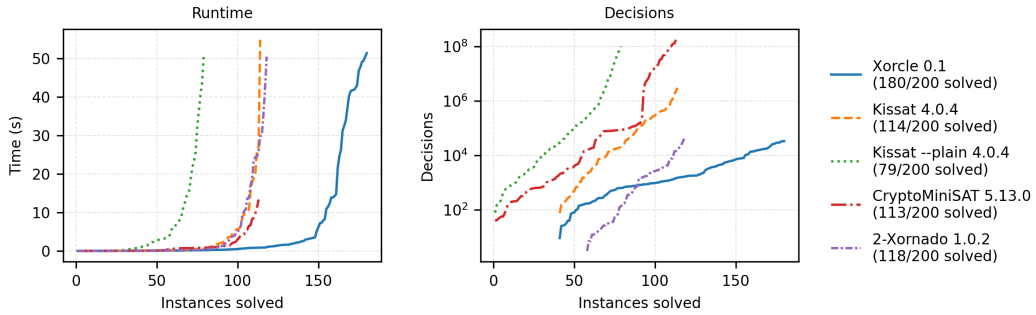

Figure 1 Cactus plot of 40 random instances from each family. Each solver is using default configurations except for Kissat `--plain`. Note that Xorcle, default Kissat, and 2-Xornado can solve some instances with 0 decisions (by unit propagation or preprocessing), so the corresponding lines on the decisions plot do not start from 0 instances solved.

Comparison with existing solvers

We benchmark Xorcle 0.1 against Kissat 4.0.4, Kissat 4.0.4 with `--plain` (disabling advanced techniques), CryptoMiniSAT 5.13.0, and 2-Xornado 1.0.2,[1] all using default configurations except for Kissat `--plain`. We convert XNF files to CNF-XOR for CryptoMiniSAT with an extension variable for each equation, then to CNF for Kissat with standard Tseitin encodings. For 2-Xornado, all formulas are converted into 2-XNF using extension variables, following the algorithm in [1]. Note that runtime errors occurred with 2-Xornado on some pebbling formulas and Tseitin formulas; these were recorded as timeouts.

Randomly sampling 40 benchmarks from each family, Xorcle's performance significantly exceeds every other solver that we tested, shown in Figure 1. In addition, Xorcle achieves this using significantly fewer decisions, indicating that Xorcle makes better quality decisions and deductions on these XNF families compared to other solvers.

We make special note of Xorcle's performance on two families. First, as shown in Figure 2, Xorcle performs exceptionally well on 2-XNF Ascon-128 instances, even beating the default configuration of the dedicated 2-XNF solver 2-Xornado. This is noteworthy because Xorcle is *not* optimized for the 2-XNF input format, using general XNF reasoning.

Second, without any preprocessing and using only core reasoning, Xorcle empirically solves CNF Tseitin formulas on expander graphs in our test set with nearly polynomially scaling running time. This is shown in Figure 3, and is noteworthy because Resolution-based solvers must take exponential time. Note that CryptoMiniSAT can trivialize Tseitin formulas through preprocessing, but by default can only recognize parities with up to 7 variables.

7 Proof Logging with LRUP(⊕)

Many SAT solvers output machine-checkable proofs for unsatisfiable instances in DRAT format [56], but DRAT does not natively support parity reasoning. CryptoMiniSAT outputs proofs in the XLRUP format [52]. This supports parity reasoning but not extension variables, which are needed to compile our reasoning about linear clauses down to CNF-XOR. Another approach uses pseudo-Boolean solving and VeriPB [24], but is more technically involved.

Xorcle outputs a new variation of the LRUP proof format [18] that we call LRUP(⊕), identical to LRUP except with parity expressions written in place of literals, with similar syntax to the XNF extension to the DIMACS input format in [1]. LRUP(⊕) proofs can be

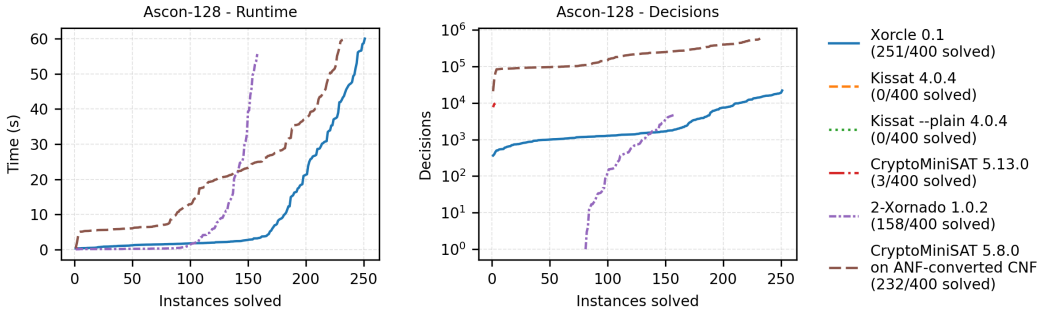

Figure 2 Cactus plot for all 400 instances from the Ascon-128 family of benchmarks. We include an extra curve to better compare with the experiment in [1]. In particular, [1] measures the performance of CryptoMiniSAT using v5.8.0 and running on CNF converted directly from ANF files (whereas we normally convert from XNF for consistency with other tests). Both changes are needed for reasonable performance; in particular, the default configuration of v5.13.0 surprisingly performs significantly worse than v5.8.0 on this family.

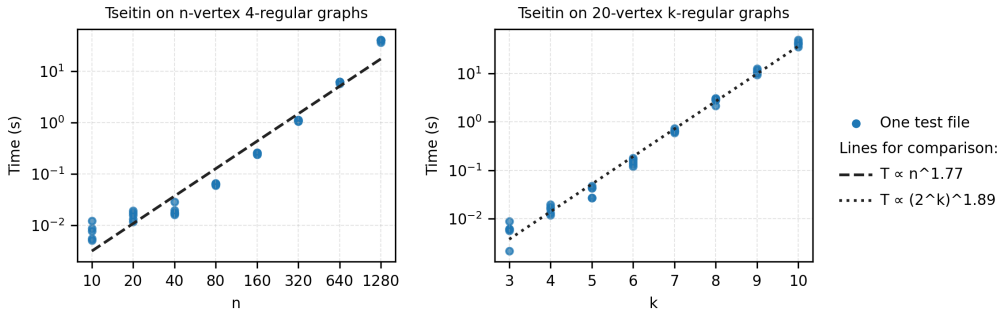

Figure 3 Xorcle's performance on Tseitin formulas for random k-regular n-vertex graphs, with 5 graphs per size, encoded in CNF. Over the tested range of inputs, scaling in k appears polynomial in the input size, while scaling in n appears to slightly exceed polynomial time.

generated by simply recording a trace of the clauses used and derived during conflict analysis, e.g. Algorithm 3. We provide a basic independently-written LRUP(⊕) proof checker that simply runs unit propagation to verify the proof.

8 Conclusion

We have demonstrated the potential and effectiveness of CDCL(⊕), through both theoretical p-simulation of the Res(⊕) proof system and empirical analysis of our proof-of-concept Xorcle on several benchmarks. Still, there are limitations to our present understanding.

Xorcle does not yet fully implement a variety of other existing CDCL heuristics, such as restarts [38, 27], clause deletion [2], clause minimization [50], preprocessing and inprocessing techniques (see [9] for a survey), and changing heuristics dynamically based on a schedule or other observations [8, 41, 14]. These remain important open directions that could lead to performance improvements on both CNF and XNF formulas.

Additionally, it would be interesting to explore heuristics that specifically target CNF formulas. For example, one might imagine swapping between Clause VSIDS and Variable VSIDS if the instance is suspected to require less parity reasoning, or using a different strategy to select an equation from a given clause in order to solve CNF problems like the bijective pigeonhole principle that require more complicated parity branching. The goal of this is not to substitute for general-purpose CDCL SAT solvers but, instead, to allow Xorcle to be useful for a wider range of CNF formulas where there is an implicit need for parity reasoning.

On the theoretical side, while Algorithm 3 is clearly algorithmically similar to the standard 1-UIP algorithm, it is less clear if the output satisfies some notion of being specifically "1-UIP" (beyond being generally asserting), relating to classical notions of unique implication points and the 1-UIP cut. Clause minimization is another classical CDCL notion typically defined with implication graphs [50]. We would also be interested to semantically characterize this and faithfully generalize it to CDCL(⊕).

References

1 Bernhard Andraschko, Julian Danner, and Martin Kreuzer. SAT solving using XOR-OR-AND normal forms. *Math. Comput. Sci.*, 18(4):20, 2024. doi:10.1007/S11786-024-00594-X.

2 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009*, pages 399–404, 2009. URL: http://ijcai.org/Proceedings/09/Papers/074.pdf.

3 Gregory V. Bard. *Algebraic Cryptanalysis*. Springer, 2009. doi:10.1007/978-0-387-88757-9.

4 Peter Baumgartner and Fabio Massacci. The taming of the (X)OR. In *Computational Logic - CL 2000*, volume 1861 of *Lecture Notes in Computer Science*, pages 508–522. Springer, 2000. doi:10.1007/3-540-44957-4_34.

5 Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res.*, 22:319–351, 2004. doi:10.1613/JAIR.1410.

6 Paul Beame and Glenn Sun. Extending cdcl to disjunctions of parity equations, 2026. arXiv:2605.15002.

7 Olaf Beyersdorff and Benjamin Böhm. Understanding the relative strength of QBF CDCL solvers and QBF resolution. *Log. Methods Comput. Sci.*, 19(2), 2023. doi:10.46298/LMCS-19(2:2)2023.

8 Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2008*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer, 2008. doi:10.1007/978-3-540-79719-7_4.

9 Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 391–435. IOS Press, 2021. doi:10.3233/FAIA200992.

10 Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. In *Proceedings of the 40th Design Automation Conference, DAC 2003*, pages 830–835. ACM, 2003. doi:10.1145/775832.776041.

11 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013*, volume 8124 of *Lecture Notes in Computer Science*, pages 200–216. Springer, 2013. doi:10.1007/978-3-642-40627-0_18.

12 Chin-Liang Chang. The unit proof and the input proof in theorem proving. *J. ACM*, 17(4):698–707, 1970. doi:10.1145/321607.321618.

13 Jingchao Chen. Building a hybrid SAT solver via conflict-driven, look-ahead and XOR reasoning techniques. In *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 298–311. Springer, 2009. doi:10.1007/978-3-642-02777-2_29.

14 Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux. Combining VSIDS and CHB using restarts in SAT. In *27th International Conference on Principles and Practice of Constraint Programming, CP 2021*, volume 210 of *LIPIcs*, pages 20:1–20:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CP.2021.20.

15 Vasek Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discret. Math.*, 4(4):305–337, 1973. doi:10.1016/0012-365X(73)90167-2.

16 Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *J. Symb. Log.*, 44(1):36–50, 1979. doi:10.2307/2273702.

17 William J. Cook, Collette R. Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discret. Appl. Math.*, 18(1):25–38, 1987. doi:10.1016/0166-218X(87)90039-4.

18 Luís Cruz-Filipe, João Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017*, volume 10205 of *Lecture Notes in Computer Science*, pages 118–135, 2017. doi:10.1007/978-3-662-54577-5_7.

19 Julian Danner. *SAT Solving Using XOR-OR-AND Normal Forms and Cryptographic Fault Attacks*. PhD thesis, University of Passau, Germany, 2025. URL: https://opus4.kobv.de/opus4-uni-passau/frontdoor/index/index/docId/1917.

20 Julian Danner and Martin Kreuzer. Conflict-driven SAT solving using XOR-OR-AND normal forms. Presentation-only talk at Pragmatics of SAT 2025, PoS 2025. URL: https://ceur-ws.org/Vol-4008/.

21 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3_37.

22 Nessan Fitzmaurice. bit: C++ header-only library for working in bit-space/GF(2), 2026. URL: https://github.com/nessan/bit.

23 Roman Gershman and Ofer Strichman. Haifasat: A new robust SAT solver. In Shmuel Ur, Eyal Bin, and Yaron Wolfsthal, editors, *Hardware and Software Verification and Testing, First International Haifa Verification Conference, HVC 2005*, volume 3875 of *Lecture Notes in Computer Science*, pages 76–89. Springer, 2005. doi:10.1007/11678779_6.

24 Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-boolean proofs. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021*, pages 3768–3777. AAAI Press, 2021. doi:10.1609/AAAI.V35I5.16494.

25 Evguenii I. Goldberg and Yakov Novikov. BerkMin: A fast and robust SAT-solver. In *2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002)*, pages 142–149. IEEE Computer Society, 2002. doi:10.1109/DATE.2002.998262.

26 Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *Proceedings of the 21st National Conference on Artificial Intelligence, AAAI 2006*, pages 54–61. AAAI Press, 2006. URL: http://www.aaai.org/Library/AAAI/2006/aaai06-009.php.

27 Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence, AAAI 1998*, pages 431–437. AAAI Press / The MIT Press, 1998. URL: http://www.aaai.org/Library/AAAI/1998/aaai98-061.php.

28 Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958. doi:10.1090/s0002-9904-1958-10224-4.

29 Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985. doi:10.1016/0304-3975(85)90144-6.

30 Jan Horácek. *Algebraic and Logic Solving Methods for Cryptanalysis*. PhD thesis, University of Passau, Germany, 2020. URL: https://opus4.kobv.de/opus4-uni-passau/frontdoor/index/index/docId/773.

31 Jan Horácek and Martin Kreuzer. Refutation of products of linear polynomials. In *Proceedings of the 3rd Workshop on Satisfiability Checking and Symbolic Computation, SC-Square@FLOC 2018*, volume 2189 of *CEUR Workshop Proceedings*, page 33. CEUR-WS.org, 2018. URL: https://ceur-ws.org/Vol-2189/paper9.pdf.

32 Dmitry Itsykson and Dmitry Sokolov. Lower bounds for splittings by linear combinations. In *Mathematical Foundations of Computer Science 2014, MFCS 2014*, volume 8635 of *Lecture Notes in Computer Science*, pages 372–383. Springer, 2014. doi:10.1007/978-3-662-44465-8_32.

33 Dmitry Itsykson and Dmitry Sokolov. Resolution over linear equations modulo two. *Ann. Pure Appl. Log.*, 171(1), 2020. doi:10.1016/J.APAL.2019.102722.

34 Tero Laitinen, Tommi A. Junttila, and Ilkka Niemelä. Extending clause learning DPLL with parity reasoning. In *ECAI 2010 - 19th European Conference on Artificial Intelligence*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 21–26. IOS Press, 2010. URL: http://www.booksonline.iospress.nl/Content/View.aspx?piid=17707.

35 Tero Laitinen, Tommi A. Junttila, and Ilkka Niemelä. Extending clause learning SAT solvers with complete parity reasoning. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012*, pages 65–72. IEEE Computer Society, 2012. doi:10.1109/ICTAI.2012.18.

36 Massimo Lauria, Jan Elffers, Jakob Nordström, and Marc Vinyals. Cnfgen: A generator of crafted benchmarks. In *Theory and Applications of Satisfiability Testing - SAT 2017*, volume 10491 of *Lecture Notes in Computer Science*, pages 464–473. Springer, 2017. doi:10.1007/978-3-319-66263-3_30.

37 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential recency weighted average branching heuristic for SAT solvers. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI 2016*, pages 3434–3440. AAAI Press, 2016. doi:10.1609/AAAI.V30I1.10439.

38 Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. In *Second Israel Symposium on Theory of Computing Systems, ISTCS 1993*, pages 128–133. IEEE Computer Society, 1993. doi:10.1109/ISTCS.1993.253477.

39 Norbert Manthey, Marijn Heule, and Armin Biere. Automated reencoding of boolean formulas. In *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012*, volume 7857 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2012. doi:10.1007/978-3-642-39611-3_14.

40 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 2001*, pages 530–535. ACM, 2001. doi:10.1145/378239.379017.

41 Chanseok Oh. Between SAT and UNSAT: the fundamental difference in CDCL SAT. In *Theory and Applications of Satisfiability Testing - SAT 2015*, volume 9340 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 2015. doi:10.1007/978-3-319-24318-4_23.

42 Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing - SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007. doi:10.1007/978-3-540-72788-0_28.

43 Knot Pipatsrisawat and Adnan Darwiche. A new clause learning scheme for efficient unsatisfiability proofs. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008*, pages 1481–1484. AAAI Press, 2008. URL: http://www.aaai.org/Library/AAAI/2008/aaai08-243.php.

44 Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011. doi:10.1016/J.ARTINT.2010.10.002.

45 Hossein M. Sheini and Karem A. Sakallah. Pueblo: A hybrid pseudo-boolean SAT solver. *J. Satisf. Boolean Model. Comput.*, 2(1-4):165–189, 2006. doi:10.3233/SAT190020.

46 Mate Soos. Enhanced gaussian elimination in DPLL-based SAT solvers. In *POS-10. Pragmatics of SAT*, volume 8 of *EPiC Series in Computing*, pages 2–14. EasyChair, 2010. doi:10.29007/G7SS.

47 Mate Soos, Stephan Gocht, and Kuldeep S. Meel. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *Computer Aided Verification - 32nd International Conference, CAV 2020*, volume 12224 of *Lecture Notes in Computer Science*, pages 463–484. Springer, 2020. doi:10.1007/978-3-030-53288-8_22.

48 Mate Soos and Kuldeep S. Meel. BIRD: Engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*, pages 1592–1599. AAAI Press, 2019. doi:10.1609/AAAI.V33I01.33011592.

49 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009. doi:10.1007/978-3-642-02777-2_24.

50 Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009. doi:10.1007/978-3-642-02777-2_23.

51 Glenn Sun. Xorcle. Software, version 0.1., swhId: swh:1:snp:4db00879b68ea0da4cb4079628e526b4c21121bc (visited on 2026-07-01). URL: https://github.com/glenn-sun/xorcle, doi:10.4230/artifacts.26951.

52 Yong Kiam Tan, Jiong Yang, Mate Soos, Magnus O. Myreen, and Kuldeep S. Meel. Formally certified approximate model counting. In *Computer Aided Verification - 36th International Conference, CAV 2024*, volume 14681 of *Lecture Notes in Computer Science*, pages 153–177. Springer, 2024. doi:10.1007/978-3-031-65627-9_8.

53 Grigori S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, 1983. English translation of 1968 original. doi:10.1007/978-3-642-81955-1_28.

54 Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, 1987. doi:10.1145/7531.8928.

55 Marc Vinyals, Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht, and Jakob Nordström. In between resolution and cutting planes: A study of proof systems for pseudo-boolean SAT solving. In *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 292–310. Springer, 2018. doi:10.1007/978-3-319-94144-8_18.

56 Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing - SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014. doi:10.1007/978-3-319-09284-3_31.