Conflict-Driven SAT Solving using XOR-OR-AND Normal Forms

JULIAN DANNER, University of Passau, Germany and TNG Technology Consulting GmbH, Germany
MARTIN KREUZER*, University of Passau, Germany

Background: Classical solvers for Boolean polynomial systems are usually based on the Conjunctive Normal Form (CNF) and the resolution calculus, or on the algebraic normal form (ANF) and the polynomial calculus, i.e., on Gröbner basis methods. Here we develop a new solver which is based on the XOR-OR-AND normal form (XNF) and the XLIN proof system. XNF formulae allow compact encodings of XOR-rich problems, which, for instance, occur naturally in cryptography.
Objectives: The paper has the following goals: lay a solid complexity-theoretic foundation for the new propositional logic proof system XLIN, devise XLIN-based conflict learning methods for formulae in XNF, create a variant of this framework optimized for practical efficiency, implement it in an actual solver called Xorricane, and finally apply this solver to benchmark suites consisting of random examples and cryptographic examples to prove its viability.
Methods: As a first step, we show that the new propositional logic proof system XLIN is polynomially equivalent equivalent to the well-studied proof systems RES($\oplus$) and SRES. It follows that there is an exponential separation between XLIN and the CNF-based resolution proof system RES. In the next step, we devise XLIN-based conflict-learning methods for formulae in XNF which are analogous to the CDCL methods of classical SAT solvers. Using Gaußian Constraint Propagation and a suitable version of conflict analysis, we obtain a conflict-driven framework CDXCL for XNF SAT-solving. To ensure practical efficiency, we also present a variant, called CDXCL$^{\text{lite}}$, which uses a weaker propagation mechanism but retains the same learning method.
Results: These theoretical advances result in Xorricane, an implementation of the latter XNF SAT-solving framework featuring novel lazy data structures. This implementation is compared to state-of-the-art algebraic and logic solving approaches on random and cryptographic benchmark suites. Although it still lacks several optimizations, it performs well on the benchmark problems and matches or outperforms modern CNF and CNF-XOR based SAT-solvers for XOR-rich problem instances.
Conclusions: Solving Boolean polynomial systems, or equivalently, SAT-solving, based on the XNF and the XLIN proof system outperforms classical methods not only theoretically, but based on the new conflict-driven XNF clause learning methods developed here, it is possible to implement an XNF solver with good practical efficiency. The advantage of XNF-based SAT-solving is strongest for XOR-rich examples such as the ones derived from cryptographic attacks.

JAIR Associate Editor: Kuldeep Meel

JAIR Reference Format:
Julian Danner and Martin Kreuzer. 2026. Conflict-Driven SAT Solving using XOR-OR-AND Normal Forms. *Journal of Artificial Intelligence Research* 86, Article 46 (August 2026), 70 pages. DOI: 10.1613/jair.1.20298

1 Introduction

One of the main areas of research in computational logic is the *SAT problem*. It asks whether there exists a satisfying assignment for a propositional logic formula given in conjunctive normal form (CNF). The SAT problem has been studied since the early 1960s as it is one of the first known NP-complete problems. The first decades were

*Corresponding Author.

Authors' Contact Information: Julian Danner, ORCID: 0009-0003-5090-0138, julian.danner@tngtech.com, University of Passau, Passau, Germany and TNG Technology Consulting GmbH, Unterföhring, Germany; Martin Kreuzer, ORCID: 0000-0002-4732-2627, martin.kreuzer@uni-passau.de, University of Passau, Passau, Germany.

DOI: 10.1613/jair.1.20298

dominated by the DPLL algorithm, which is based on a straightforward branch-and-propagate strategy, until the field gained traction with the introduction of conflict-learning methods and the CDCL framework. Since then, new techniques and ideas have constantly been added, and state-of-the-art *SAT solvers* have evolved into versatile, highly scalable, complex programs with numerous applications in industry and research. On the other hand, in computational algebra, the existence of solutions to polynomial systems is studied. It is well-known that this question is equivalent to the SAT problem for systems of *Boolean polynomials*. Interestingly, both worlds – algebra and logic – use somewhat different and sometimes even orthogonal approaches to approach this problem.

A significant disadvantage of algebraic methods is that, unlike modern logic solvers, they scale poorly with increasing problem size. Conversely, a major weakness in SAT solving is the inability to efficiently handle systems of XOR constraints. In the algebraic domain, however, XOR constraints correspond to linear equations over $\mathbb{F}_2$, which are readily solvable and straightforward to handle. Especially for instances originating from cryptographic applications, traditional SAT solvers could only contribute little initially. In the last two decades, this has changed markedly with various heuristics and preprocessing methods, as well as statistical techniques using model counters based on SAT solvers. Yet, for certain small instances, algebraic methods – such as Boolean Gröbner bases – can achieve remarkable results, and sometimes they are employed as components of mixed attack strategies. It is at hand to seek a unified algorithm allowing SAT solvers to perform algebraic reasoning natively without losing their agility and speed.

In recent years, some efforts have been made in this direction. However, instead of a full combination, they either add algebraic reasoning to existing logical solvers, see (Dreyer and Nguyen 2013; Zengler and Küchlin 2010), or move back and forth between the two worlds, see (Choo et al. 2019). A different approach was initiated in (Horáček and Kreuzer 2018) with the introduction of the propositional proof system SRES. It generalizes the *resolution* proof system RES, which is the foundation of CNF-based logical solving, to the richer language of XOR-OR-AND normal forms (XNF). The latter arises from the CNF by replacing *literals* with linear XOR constraints, called *linerals*. It provides a natural handling of XOR constraints and allows compact encodings of XOR-rich formulae. Another proof system that works with XNF clauses is RES($\oplus$) (Itsykson and Sokolov 2014). Contrary to SRES, this proof system has been researched more thoroughly in (Beame and Koroth 2023; Chattopadhyay et al. 2023; Efremenko et al. 2024; Garlík and Kołodziejczyk 2018; Gryaznov et al. 2024; Itsykson and Sokolov 2020; Khaniki 2022; Oparin 2016) with the ultimate goal of finding classes of unsatisfiable formulae with exponential lower bounds on the size of the shortest refutations. While this goal has not yet been reached, RES($\oplus$) was shown to be exponentially stronger than RES and is considered a fairly strong proof system. On the practical side, however, no RES($\oplus$)-based solvers have been developed.

To the best of our knowledge, so far there are only two XNF-based solvers. The first was introduced in (Horáček 2020) following a DPLL framework and reasoning within SRES. There is no public implementation available. The second solver, 2-Xornado, described in (Andraschko et al. 2024), also reasons within SRES. It solves XNF formulae by first reducing them in polynomial-time to 2-XNF formulae, where each XNF clause is a disjunction of at most two linerals. Then another DPLL-based framework with graph-based in- and preprocessing methods is employed. First experimental results for 2-Xornado show good performance on certain random and cryptographic benchmarks, but also indicate poor scaling properties.

This paper continues this line of research. On one hand, we introduce two new XNF-based proof systems, XLIN and XRES, and show that they are polynomially equivalent to SRES and RES($\oplus$). This essentially shows that all these proof systems are equally strong. On the other hand, we generalize the framework of conflict-driven clause learning, CDCL, to the setting of XNF formulae and call the result CDXCL. Its conflict learning method is backed by XLIN-proofs, i.e., it can be easily converted to RES($\oplus$)- and SRES-proofs. To ensure practical efficiency, we also introduce CDXCL$^{\text{lite}}$, a variant that uses a weaker propagation mechanism but the same stronger learning method. The theoretical construction is complemented with lazy data structures which allow an efficient implementation

of the latter framework. This is comparable to similar approaches in the CNF setting. Finally, to assess the practical impact, we implemented the framework CDXCL$^{\text{lite}}$ using a selection of established decision, restart, and deletion heuristics in the solver Xorricane. Despite some remaining shortcomings, our implementation outperforms several state-of-the-art algebraic and logic solving methods on cryptographic and random benchmarks. The source code is available at https://github.com/j-danner/Xorricane-paper.

Before we describe the structure and contents of our contributions in more detail, let us give a short overview of CNF-based SAT solving in the context of XOR-rich formulae.

1.1 Conflict-Driven SAT Solving

The first algorithm to construct satisfying assignments of a given formula in CNF, known as DPLL (Davis et al. 1962), approached the problem by repeatedly splitting the search space by guessing the truth value of a variable and applying *Boolean constraint propagation* (BCP) to distribute fixed truth values to the clauses. If a new assignment is derived, the corresponding clause is known as its *reason clause*. Whenever all the literals in a clause are assigned to false, the clause is known as a *conflict clause*, and the current assignment cannot satisfy the formula. Therefore the algorithm backtracks to the latest decision and inverts the corresponding assignment.

Major algorithmic progress was made in the late 1990s with the advent of *conflict analysis* in CDCL (Marques-Silva and Sakallah 1996; Moskewicz et al. 2001): whenever the algorithm encounters a conflict clause, root causes for the conflict, i.e., a (small) set of prior assignments, are determined and encoded by a new *learnt clause*. This clause can be computed from a *propagation graph* that represents the dependencies of the implied assignments. During conflict learning, a specific cut in this graph, i.e., a set of literals separating the decision variables from the conflict, forms the learnt clause. When added to the CNF formula, it enhances propagation with BCP and speeds up the search altogether.

Over the years, this general CDCL framework has seen many gradual improvements, including techniques like *restarts* (Biere and Fröhlich 2018; Gomes et al. 1998; Moskewicz et al. 2001), *dynamic decision heuristics* (Biere and Fröhlich 2015; Eén and Sörensson 2004; Moskewicz et al. 2001; Pipatsrisawat and Darwiche 2007), *phase saving* (Biere and Fröhlich 2015; Pipatsrisawat and Darwiche 2007), *clause minimization* (Beame, Kautz, et al. 2004; Eén and Sörensson 2004; Sörensson and Biere 2009), *on-the-fly subsumption* (Hamadi et al. 2009; H. Han and Somenzi 2009; Soos, Nohl, et al. 2009), *clause deletion* (Audemard and Simon 2009; Krüger et al. 2022; Oh 2015), *stable/unstable phases* (Oh 2015), *chronological backtracking* (Nadel and Ryvchin 2018), and a wide range of *in-* and *pre-processing* methods, e.g., see (Balyo et al. 2014; Biere, Heule, et al. 2021; Eén and Biere 2005; H. Han and Somenzi 2007; Heule et al. 2013; Järvisalo, Biere, et al. 2010; Järvisalo, Heule, et al. 2012; Li et al. 2020; Lynce and Marques-Silva 2003). On the practical side, modern implementations of CDCL also provide efficient, cache-friendly, lazy data structures that are a major contributor to the success of SAT solving, e.g., see (Chu et al. 2010; Gent 2013; Hölldobler et al. 2010; Lewis et al. 2005; Moskewicz et al. 2001).

The core conflict-learning methods and a majority of the above enhancements are justified by the proof system RES. In particular, for unsatisfiable CNF formulae, the solvers can produce RES-refutations, i.e., proofs of unsatisfiability, whose size is polynomial in the runtime of CDCL. Thus, exponential lower bounds on refutation sizes also induce exponential lower bounds on the runtime of CDCL. Several such hard classes of CNF formulae are known, e.g., see (Haken 1985; Razborov 2004).

1.2 Solving XOR-Rich Formulae

In applications like cryptanalysis, circuit verification, and approximate model counting, XOR-rich formulae occur naturally. These formulae are challenging for CNF-based SAT solvers (Emdin et al. 2024; Gwynne and Kullmann 2014; Li 2000; Selman et al. 1997; Warners and van Maaren 1998), as long XOR constraints cannot be

encoded in CNF without loosing their linear structure (Emdin et al. 2024; Gwynne and Kullmann 2014). Different techniques to overcome this issue have been designed.

One line of research investigates CNF-XOR formulae, which consist of a CNF formula and a set of XOR clauses, i.e., XOR-chains of literals. In (C.-S. Han and Jiang 2012; Laitinen et al. 2012a,b, 2011, 2010, 2012c, 2013; Soos 2010; Soos, Gocht, et al. 2020; Soos and Meel 2019; Soos, Nohl, et al. 2009) an adaptation of the CDCL framework to CNF-XOR formulae is studied. The propagation method is extended to work with XOR clauses, and denoted here as $\mathrm{BCP}^{\mathrm{XOR}}_{\mathrm{GJE}}$. It uses BCP for the CNF part of the formula and a Gauß-Jordan mechanism for the XOR constraints. The conflict learning is adapted to weaken any XOR clauses to corresponding CNF clauses whenever they are used during conflict analysis. We denote this generalization of CDCL to CNF-XOR formulae by $\mathrm{CDCL}^{\mathrm{XOR}}_{\mathrm{GJE}}$. The implementations resulting from this research are spearheaded by the CNF-XOR SAT solver CryptoMiniSat (Soos, Nohl, et al. 2009). Although this approach allows shorter problem encodings due to the richer language of CNF-XOR and the stronger propagation mechanism, the conflict learning still happens on the CNF level within the proof system RES, and the above theoretical limitations still apply.

To counteract this, one may use logic and algebraic solvers in parallel, as suggested in (Horáček, Burchard, et al. 2017), or sequentially, as developed in (Choo et al. 2019; Dreyer and Nguyen 2013; Zengler and Küchlin 2010). All these techniques, however, have not been able to balance the speed of logic solving with the power of algebraic reasoning well enough to be worthwhile.

1.3 Structure and Contents

In search of a better trade-off, in this paper we stick strictly to the core logical framework of CDCL, but use the reasoning power of the stronger proof system XLIN. In particular, we work with the more expressive language of XNFs, generalizing both CNF and CNF-XOR.

Throughout the paper we use an algebraic language. To this end, we let

$$\mathbb{B}_n \;=\; \mathbb{F}_2[x_1,\dots,x_n] \,/\, \langle x_1^2 + x_1, \dots, x_n^2 + x_n\rangle$$

be the Boolean polynomial ring in n indeterminates and denote the $\mathbb{F}_2$-vector subspace containing the linear Boolean polynomials by $\mathbb{L}_n \;=\; \langle x_1,\dots,x_n,1\rangle_{\mathbb{F}_2}$. Then *linerals*, i.e., XORs of literals, correspond to linear Boolean polynomials. Furthermore, XNF clauses correspond to products of linear polynomials and, thereby, to subsets of $\mathbb{L}_n$. Altogether, an XNF formula is identified by a subset of the power set $\mathcal{P}(\mathbb{L}_n)$ of $\mathbb{L}_n$, see Notation 2.5.

Dealing with XNF clauses introduces a few complications which can be traced back to the following simple equality for $\ell_1, \ell_2 \in \mathbb{L}_n$:

$$\ell_1 \cdot \ell_2 \;=\; \ell_1 \cdot (\ell_1 + \ell_2 + 1).$$

It shows that there are distinct clauses $C_1, C_2 \subseteq \mathbb{L}_n$ corresponding to the same Boolean polynomial, i.e., with $\prod C_1 = \prod C_2$. Therefore, they also have the same set of satisfying assignments and are equivalent as logical formulae, i.e., we have $C_1 \equiv C_2$. To avoid issues with different representations of XNF clauses which encode essentially the same information, we provide the following useful characterization of equivalent XNF clauses in Section 2. To state it, we introduce the *associated subspace* of a given clause $C \subseteq L$ as the $\mathbb{F}_2$-vector subspace $V_C \;=\; \langle 1 + C\rangle_{\mathbb{F}_2} \;=\; \langle \ell + 1 \mid \ell \in C\rangle_{\mathbb{F}_2}$ of $\mathbb{L}_n$. In Proposition 2.12 we prove that for two clauses $C_1, C_2 \subseteq \mathbb{L}_n$ we have

$$C_1 \;\equiv\; C_2 \quad\Longleftrightarrow\quad V_{C_1} \;=\; V_{C_2} \text{ or } 1 \in V_{C_1} \cap V_{C_2}.$$

In particular, a clause $C \subseteq \mathbb{L}_n$ is a tautology if and only if $1 \in V_C$. Methods from linear algebra can now decide the equivalence of XNF clauses in polynomial time.

In Section 3, we offer some first steps towards delineating and understanding the theoretical limits of SRES- and RES($\oplus$)-based solvers by providing several polynomially equivalent formulations. In particular, we introduce the proof systems XLIN and XRES as further natural extensions of RES and RLIN (Horáček 2020). After studying polynomial simulations of their rules of inference in Proposition 3.10, we show that in fact all four XNF-based proof

systems are polynomially equivalent in Proposition 3.11. Thus, theoretical results on exponential lower bounds of minimal refutation sizes for RES(⊕) can be translated to SRES and XLIN, showing them to be exponentially stronger than RES. Therefore solvers based on any of these proof systems and using an analogue of CDCL might be able to speed up solving times considerably.

With this in mind, we properly investigate conflict-driven XNF SAT solving algorithms in Sections 4 and 5. As a replacement for the propagation mechanism BCP, we use *Gaußian Constraint Propagation* (GCP) as introduced in (Horáček 2020). Where BCP only assigns variables to true or false during propagation, GCP uses all available linear information to replace variables by linerals, i.e., linear polynomials. This makes GCP a rather strong propagation method. As for the conflict learning, one can construct a *propagation graph* analogous to the CNF setting, which captures the execution of GCP. Using cuts in this graph, one can also learn clauses as desired. This, however, does not work out as efficiently as hoped for, due to the following obstruction. In the CNF setting, each vertex and its in-neighbours correspond to a reason clause. However, for XNF formulae, the vertices and their in-neighbours form only *weakened* versions of the actual reason XNF clauses. This can be attributed to the fact that multiple linerals may need to be combined to imply the negation of a lineral in a given reason clause. These weakened clauses generally contain more linerals. Consequently, the utility of such learnt XNF clauses is significantly diminished.

To overcome this impediment, we look at the logical reasoning behind the cuts in the propagation graph. In the CNF setting, they follow from the iterated computation of *resolvents* along the sequence of reason clauses which are used during propagation with BCP. By mimicking these computations with the corresponding reason XNF clauses used during propagation with GCP, it is indeed possible to learn new XNF clauses. This, however, makes it necessary to carefully rewrite the affected clauses before applying the rule of inference. Our description avoids an explicit change of representation altogether by formulating the resolvent computations on the respective *equivalence classes*, i.e., using their associated subspaces.

In this context, we introduce effective characterizations of *conflict* and *reason* clauses in Proposition 4.8. They state that, under a set of linerals $L \subseteq \mathbb{L}_n$, a clause $C \subseteq \mathbb{L}_n$ is a conflict clause if and only if $V_C \subseteq \langle L \rangle_{\mathbb{F}_2}$, and $C \subseteq \mathbb{L}_n$ is a non-trivial reason clause if and only if $\dim_{\mathbb{F}_2}\left(V_C / V_C \cap \langle L \rangle_{\mathbb{F}_2}\right) = 1$.

The computation of resolvents is then substituted by Proposition 4.37 which can be paraphrased as follows. Suppose that $L \subseteq \mathbb{L}_n$ is the set of linerals propagated with GCP, and we have XNF clauses $C_1, C_2 \subseteq \mathbb{L}_n$ which are reason clauses under L for $\ell_1 \in \mathbb{L}_n$ and $\ell_2 \in \mathbb{L}_n$. Then there is a reason clause $R \subseteq \mathbb{L}_n$ for $\ell_1 + \ell_2$ under L with a polynomial-size XLIN-proof from C_1, C_2 and

$$V_R \;=\; \left(V_{C_1} \cap \langle L \rangle_{\mathbb{F}_2}\right) \;+\; \left(V_{C_2} \cap \langle L \rangle_{\mathbb{F}_2}\right) \;+\; \langle \ell_1 + \ell_2 + 1 \rangle_{\mathbb{F}_2}.$$

Now we can proceed as in the CNF setting by processing the XNF reason clauses in the order dictated by GCP and by computing the associated subspaces V_R in each step via the above formula. This effectively eliminates the dependency of V_R on all linerals found by GCP one-by-one, until the associated XNF clause R does not depend on any linerals from the highest decision level, i.e., until R is an *asserting clause*. The corresponding algorithm is denoted by LinTrailRes and all its reasoning steps are backed by polynomial-size XLIN-proofs.

Finally, we combine GCP with LinTrailRes in Algorithm 4 (CDXCL) to obtain the desired generalization of CDCL (and $\mathrm{CDCL}^{\mathrm{XOR}}_{\mathrm{GJE}}$) to XNF formulae. Even though it is provably better than CDCL, practical implementations are hard to achieve. Not only is propagation with GCP memory-heavy, also the numerous intersections of vector subspaces that need to be computed during an execution of LinTrailRes prevent a linear runtime, see Remark 4.44.

To allow for efficient implementations, we offer a simplified propagation method in $\mathrm{GCP}^{\mathrm{lite}}$ (see Algorithm 5). It differs from GCP by propagating only variables which are set to true or false to the XNF clauses. While similar to BCP, here we deduce *linerals* from reason clauses. To leverage this linear information, we suggest two post-processing methods for newly derived linerals in Remark 5.7. In combination with the same strong conflict-learning method LinTrailRes, this yields $\mathrm{CDXCL}^{\mathrm{lite}}$.

Inspired by the lazy CNF-based data structures from (Moskewicz et al. 2001) and (C.-S. Han and Jiang 2012), we develop novel *watching structures* for an efficient implementation of $\mathsf{GCP}^{\mathsf{lite}}$ in Section 5.3. In particular, for each XNF clause $C \subseteq \mathbb{L}_n$, we track a generating system S of the associated subspace V_C as well as two distinct variables in distinct linerals $\ell_1, \ell_2 \in S$ which are *not* shared. Now, whenever a variable is assigned, we update the *watched variables* and the set S. If the data structures are up to date, checking whether the clause is a reason clause becomes a constant time operation, see Proposition 5.21. While the necessary changes to the generating set S seem to be a limitation at first, they actually prepare the watching structure for the conflict analysis. This is detailed in Section 5.4, where we carefully track some additional information during the update steps, such that *filtrations* (see Definition 5.37) for XNF reason clauses can be constructed. These can be converted to and from watching structures with only a small overhead. Most importantly, they can be combined efficiently, allowing a fast computation of the subspace intersections during LinTrailRes.

On the practical side, we test the efficacy of this new approach with a C++ implementation of $\mathsf{CDXCL}^{\mathsf{lite}}$, called Xorricane. It uses a small range of established heuristics and techniques from CNF-based SAT solving such as an EVSIDS *decision heuristic* (Biere and Fröhlich 2015; Eén and Sörensson 2004; Moskewicz et al. 2001; Pipatsrisawat and Darwiche 2007) with *phase saving* (Pipatsrisawat and Darwiche 2007), *dynamic restart scheduling* (Audemard and Simon 2009; Biere and Fröhlich 2018), and a (tier-based) *clause deletion heuristic* (Oh 2015). However, compared to modern CNF-based solvers, it has no *chronological backtracking*, no *clause minimization*, and offers only one in- and pre-processing method. Moreover, the implementation is not cache-optimized and the parameters for the employed heuristics were only fine-tuned on a small benchmark set with a short timeout. Implementation details can be found in Appendix A and the source code and benchmark instances are publicly available at https://github.com/j-danner/Xorricane-paper.

In Section 6 we evaluate the performance of Xorricane on random and cryptographic benchmark suites. The cryptographic benchmark suites consist of instances related to simplified state- and key-recovery attacks on the stream and block ciphers Ascon (Dobraunig et al. 2021), Bivium (Maximov and Biryukov 2007), and CTC2 (Courtois 2007a). Despite its early state of development, Xorricane outperforms many algebraic and CNF-based logic solving techniques, like PolyBoRi (Brickenstein and Dreyer 2009), Bosphorus (Choo et al. 2019), Glucose (Audemard and Simon 2009), CryptoMiniSat (Soos, Nohl, et al. 2009) on structured cryptographic instances. It is almost on par with an older version of the latter solver, CryptoMiniSat 2.4, and is only beat by the winner of the SAT Competition 2024, Kissat (Biere, Fazekas, et al. 2020), a modern cache-optimized solver with a broad range of in- and preprocessing methods. Most notably, for two of the three cryptographic benchmark suites, XNF-based methods prove to be the best choice.

2 XNF Clauses and Associated Subspaces

In the following we use the definitions and notations given in (Andraschko et al. 2024), (Kreuzer and Robbiano 2000), and (Krajiůek 2019). In particular, given $n \in \mathbb{N}$, we let $X_1, \dots, X_n$ be logical variables and $x_1, \dots, x_n$ be polynomial indeterminates, and we let σ be a term ordering. Since this ordering is only used with linear terms, it suffices to view it as a linear ordering of $x_1, \dots, x_n$.

Furthermore, we denote the Boolean polynomial ring by

$$\mathbb{B}_n \;=\; \mathbb{F}_2[x_1, \dots, x_n] \,/\, \langle x_1^2 + x_1, \dots, x_n^2 + x_n \rangle,$$

the subspace of linear Boolean polynomials by $\mathbb{L}_n \;=\; \langle x_1, \dots, x_n, 1 \rangle_{\mathbb{F}_2}$, the *support* of $f \in \mathbb{B}_n$ by $\mathrm{Supp}(f)$, and the *leading term* of $f \in \mathbb{B}_n$ with respect to σ by $\mathrm{LT}_\sigma(f) = \max_\sigma \mathrm{Supp}(f)$. Additionally, the *set of common zeros* of $S \subseteq \mathbb{B}_n$ is denoted by

$$\mathcal{Z}(S) \;=\; \{a \in \mathbb{F}_2^n \mid f(a) = 0 \text{ for all } f \in S\}$$

and the *set of satisfying assignments* of a logical formula F in n variables by

$$\mathcal{S}(F) \;=\; \{a \in \mathbb{F}_2^n \mid F(a) = 1\},$$

where we identify 1 with true and 0 with false. Recall that, for logical formulae F, G, we say that F *semantically implies* G if $\mathcal{S}(F) \subseteq \mathcal{S}(G)$. In this case we write $F \models G$.

Throughout the paper we work with propositional logic formulae of the following form.

Definition 2.1. Let $X_1, \ldots, X_n$ be logical variables, and let $k, m \in \mathbb{N}$.

(a) A **lineral** is a XOR chain of logical variables $X_{i_1} \oplus \cdots \oplus X_{i_m}$ or its negation $\neg(X_{i_1} \oplus \cdots \oplus X_{i_m})$, where $i_1, \ldots, i_m \in \{1, \ldots, n\}$ are pairwise distinct.

(b) An **(XNF) clause** is a formula F of the form $L_1 \vee \cdots \vee L_k$, where the L_i are linerals.

(c) A formula $F = C_1 \wedge \cdots \wedge C_m$ is in **XOR-OR-AND normal form (XNF)** if $C_1, \ldots, C_m$ are XNF clauses.

(d) A formula F is in k-**XNF** if every clause C of F involves at most k linerals.

Remark 2.2. Every literal, i.e., a variable or its negation, is a lineral with exactly one variable. Hence every formula in CNF is also in XNF. Moreover, by definition, every XOR clause is equivalent to a lineral. This means that 1-XNF clauses correspond to XOR clauses, and thus the XNF also generalizes the CNF-XOR representation. Informally speaking, we have the following chain of containments:

$$\mathsf{CNF} \;\subseteq\; \mathsf{CNF\text{-}XOR} \;\subseteq\; \mathsf{XNF}.$$

In particular, the SAT problem for XNF formulae, simply called XNF SAT, is also NP-complete. From here on we switch to an algebraic point of view. It uses the following concept of *algebraic representations*.

Definition 2.3. Let F be a propositional logic formula in n variables. Then the unique ideal $I \subseteq \mathbb{B}_n$ with $\mathcal{S}(F) = \mathcal{Z}(I)$ is called the **algebraic representation** of F.

Using this identification XOR essentially corresponds to *addition* and OR corresponds to *multiplication* in $\mathbb{B}_n$. Altogether, XNF formulae correspond to ideals generated by products of linear Boolean polynomials.

Remark 2.4 (Algebraic Representation of XNF Clauses and Formulae)**.**

(a) The algebraic representation of the literal X_i is the ideal $\langle x_i + 1\rangle$, and the algebraic representation of the literal $\neg X_i$ is the ideal $\langle x_i\rangle$. Consequently, linerals correspond to linear Boolean polynomials, e.g., $X_1 \oplus X_2$ corresponds to $\langle x_1 + x_2 + 1\rangle$.

(b) Let $C = L_1 \vee \cdots \vee L_k$ be an XNF clause with linerals $L_1, \ldots, L_k$. For $i \in \{1, \ldots, k\}$, the algebraic representation of L_i is given by the ideal $\langle \ell_i\rangle$ for some $\ell_i \in \mathbb{L}_n$. Then $\langle \ell_1 \cdots \ell_k\rangle$ is the algebraic representation of C. Thus, linerals correspond to linear Boolean polynomials, and XNF clauses correspond to products of linear Boolean polynomials.

(c) Let F be an XNF formula with clauses $C_1, \ldots, C_m$. For $i \in \{1, \ldots, r\}$, let $c_i \in \mathbb{B}_n$ be the product of linear Boolean polynomials representing C_i. Then the algebraic representation of F is the ideal $\langle c_1, \ldots, c_r\rangle$ in $\mathbb{B}_n$.

In view of this remark, linear Boolean polynomials $\ell \in \mathbb{L}_n$ are simply called *linerals*, and we identify the following descriptions of XNF formulae.

Notation 2.5. In what follows we identify the following objects with each other:

(R1) a propositional logic formula $\bigwedge_{i=1}^{m} \bigvee_{j=1}^{s_i} L_{i,j}$, where $L_{i,j}$ are linerals or constants,

(R2) a set of sets $\big\{\, \{L_{i,j} \mid j \in \{1, \ldots, s_i\}\} \mid i \in \{1, \ldots, m\} \,\big\}$, where $L_{i,j}$ are linerals or constants, and

(R3) a subset of the power set $\mathcal{P}(\mathbb{L}_n)$ of the set of linear Boolean polynomials $\mathbb{L}_n$.

This paper almost exclusively works with notation (R$_3$). Thus, a set of linear Boolean polynomials is also called an *(XNF) clause* and a subset of $\mathcal{P}(\mathbb{L}_n)$ is called an *XNF formula*. Similarly, we identify CNF clauses with subsets of $\mathbb{X}_n = \{x_1, \dots, x_n\} \cup \{x_1 + 1, \dots, x_n + 1\}$, CNF formulae with subsets of $\mathcal{P}(\mathbb{X}_n)$, and CNF-XOR formulae with subsets of $\mathcal{P}(\mathbb{X}_n) \cup \{\{\ell\} \mid \ell \in \mathbb{L}_n\}$.

In that regard, we also denote the set of satisfying assignments of an XNF clause $C \subseteq \mathbb{L}_n$ by $\mathcal{S}(C)$. Additionally, for sets $S \subseteq \mathbb{B}_n$, we abbreviate $\prod_{f \in S} f$ by $\prod S$.

Example 2.6. Let F be the XNF formula $\neg X_1 \vee \neg X_2 \vee (\neg X_1 \oplus X_3)$ consisting of a single XNF clause. Then F is also represented by the set $C = \{x_1, x_2, x_1 + x_2\} \subseteq \mathbb{L}_2$ with the following algebraic representation

$$\langle \textstyle\prod C \rangle = \langle x_1 \cdot x_2 \cdot (x_1 + x_2) \rangle = \langle x_1^2 x_2 + x_1 x_2^2 \rangle = \langle x_1 x_2 + x_1 x_2 \rangle = \langle 0 \rangle.$$

This means that $\mathcal{S}(F) = \mathbb{F}_2^2$, i.e., $F \equiv \top$ is a tautology.

In the remaining part of this section, we take a closer look at XNF clauses and examine when two clauses encode the same information. As a side-effect, this also allows us to quickly determine if an XNF clause is a tautology.

Remark 2.7. Let $C_1, C_2 \subseteq \mathbb{L}_n$ be two XNF clauses. From Remark 2.4.a and the fact that Boolean polynomial ideals are uniquely determined by their set of zeros, we obtain

$$C_1 \equiv C_2 \iff \mathcal{S}(C_1) = \mathcal{S}(C_2) \iff \mathcal{Z}(\textstyle\prod C_1) = \mathcal{Z}(\prod C_2) \iff \prod C_1 = \prod C_2.$$

Thus, the equivalence of XNF clauses can be checked by computing $\prod C_1$ and $\prod C_2$.

Since expanding these products can lead to polynomials with exponentially many terms, we strive for a faster method to decide whether $\prod C_1 = \prod C_2$. As a starting point, let us first look at the complement of the set of zeros of an XNF clause. Recall that we denote the set $\{1 + \ell \mid \ell \in C\}$ by $1 + C$.

Remark 2.8.

(a) Let $f, g_1, \dots, g_s \in \mathbb{B}_n$. For all $a \in \mathbb{F}_2^n$, we have $f(a) \neq 0$ if and only if $f(a) = 1$. Thus, $f = g_1 \cdots g_s$ implies

$$\begin{aligned}\mathcal{Z}(f + 1) &= \mathbb{F}_2^n \setminus \mathcal{Z}(f) = \mathbb{F}_2^n \setminus \textstyle\bigcup_{i=1}^n \mathcal{Z}(g_i) = \bigcap_{i=1}^n \big(\mathbb{F}_2^n \setminus \mathcal{Z}(g_i)\big) \\ &= \textstyle\bigcap_{i=1}^n \mathcal{Z}(g_i + 1) = \mathcal{Z}(\langle g_1 + 1, \dots, g_s + 1 \rangle).\end{aligned}$$

This implies $\langle f + 1 \rangle = \langle g_1 + 1, \dots, g_s + 1 \rangle$.

Conversely, $\langle f + 1 \rangle = \langle g_1 + 1, \dots, g_s + 1 \rangle$ implies $\langle f + 1 \rangle = \langle 1 + g_1 \cdots g_s \rangle$. Therefore we get $f = g_1 \cdots g_s$.

Thus $f = g_1 \cdots g_s$ if and only if $\langle f+1 \rangle = \langle g_1+1, \dots, g_s+1 \rangle$. In particular, there is a one-to-one correspondence between factorizations of f and generating sets of $\langle f + 1 \rangle$.

(b) For two XNF clauses $C_1, C_2 \subseteq \mathbb{L}_n$, Remark 2.7 and part (a) show

$$C_1 \equiv C_2 \iff \textstyle\prod C_1 = \prod C_2 \iff 1 + \prod C_1 = 1 + \prod C_2 \iff \langle 1 + C_1 \rangle = \langle 1 + C_2 \rangle.$$

Example 2.9. Let $C_1 = \{x_1 + 1,\ x_1 + x_2,\ x_1 + x_2 + x_3\} \subseteq \mathbb{L}_3$ be an XNF clause and $C_2 = \{x_1 + 1, x_2, x_3 + 1\} \subseteq \mathbb{X}_3$ be a CNF clause. Then it is easy to check that

$$\langle 1 + C_1 \rangle = \langle x_1,\ x_1 + x_2 + 1,\ x_1 + x_2 + x_3 + 1 \rangle = \langle x_1,\ x_2 + 1,\ x_3 \rangle = \langle 1 + C_2 \rangle.$$

By Remark 2.8.b, this shows that the clauses C_1 and C_2 are equivalent. The same result can be obtained with Remark 2.7, verifying that

$$\textstyle\prod C_1 = x_1 x_2 x_3 + x_1 x_2 + x_2 x_3 + x_2 = \prod C_2.$$

Observe that for an XNF clause $C \subseteq \mathbb{L}_n$, the ideal $\langle 1 + C \rangle$ of $\mathbb{B}_n$ is generated by linear polynomials. Thus we introduce the following notion.

Definition 2.10. Let $C \subseteq \mathbb{L}_n$ be an XNF clause.

(a) The ideal $I_C = \langle 1 + C \rangle$ of $\mathbb{B}_n$ is called the **associated ideal** of C.

(b) The subspace $V_C = \langle 1 + C \rangle_{\mathbb{F}_2}$ of $\mathbb{L}_n$ is called the **associated subspace** of C.

By Remark 2.8.a, the zeros of the associated subspace V_C of an XNF clause C are exactly the assignments which *violate* the clause C, i.e., $\mathcal{S}(C) = \mathbb{F}_2^n \setminus \mathcal{Z}(V_C)$. Moreover, the associated ideal of C is generated by the linear Boolean polynomials in $1 + C$, which also generate the associated subspace of C. For such ideals, we have the following useful observation.

Lemma 2.11. Let $L \subseteq \mathbb{L}_n$ be a non-empty subset, let $V = \langle L \rangle_{\mathbb{F}_2}$ be the $\mathbb{F}_2$-vector space generated by L, and let $I = \langle L \rangle$ be the ideal of $\mathbb{B}_n$ generated by L.

(a) If $1 \notin V$ then we have $V = I \cap \mathbb{L}_n$.

(b) We have $1 \in V$ if and only if $1 \in I$.

Proof. To show (a), we note that the inclusion $\subseteq$ is obviously true, whence it suffices to prove $\supseteq$. The set of zeros of I in $\mathbb{F}_2^n$ is an affine subspace of dimension $n - d$, where $d = \dim_{\mathbb{F}_2}(V)$. In particular, it consists of 2^{n-d} points. Now, if the intersection $W = I \cap \mathbb{L}_n$ contains V properly, the zero set of I is contained in the affine subspace defined by W, and $n - \dim_{\mathbb{F}_2}(W) < n - d$ is a contradiction. Thus we get $W = V$, as claimed. Part (b) follows immediately from (a). □

Using this lemma, we can characterize the equivalence of XNF clauses as follows.

Proposition 2.12. Let $C, C_1, C_2 \subseteq \mathbb{L}_n$ be XNF clauses.

(a) The following conditions are equivalent.

(1) $C_1 \equiv C_2$

(2) $I_{C_1} = I_{C_2}$

(3) $V_{C_1} = V_{C_2}$ or $1 \in V_{C_1} \cap V_{C_2}$

(b) The following conditions are equivalent.

(1) $C \equiv \{0\}$, i.e., C is a tautology.

(2) $1 \in I_C$

(3) $1 \in V_C$

Proof. The equivalence (1)⇔(2) in (a) follows from Remark 2.8.b. Next, we prove (b). If $C \equiv \{0\}$ then the above yields $I_C = I_{\{0\}} = \langle 1 \rangle$, and thus $1 \in I_C$. Now Lemma 2.11.b yields $1 \in V_C$. Finally, from $1 \in V_C$, we get $I_C = \langle 1 \rangle = I_{\{0\}}$, and hence $C \equiv \{0\}$ by what we have shown already. The implication (2)⇒(3) in (a) follows from the lemma. It remains to prove (3)⇒(2) in part (a). If $V_{C_1} = V_{C_2}$ then we have $I_{C_1} = I_{C_2}$ by definition, and if $1 \in V_{C_1} \cap V_{C_2}$ then $I_{C_1} = \langle 1 \rangle = I_{C_2}$ as well. □

This shows that the equivalence of XNF clauses can be checked in polynomial time using methods from linear algebra. The proposition can also be summarized as follows.

Remark 2.13. Let $\mathbb{S}_n = \left\{ \prod C \mid C \subseteq \mathbb{L}_n \right\} \subseteq \mathbb{B}_n$ be the set of all products of linear Boolean polynomials, i.e., the set of equivalence classes of XNF clauses. Then the map

$$\begin{array}{ccc} \mathbb{S}_n \setminus \{0\} & \longrightarrow & \left\{ U \subseteq \mathbb{L}_n \text{ subspace} \mid 1 \notin U \right\} \\ \prod C & \longmapsto & V_C \end{array}$$

is well-defined and bijective.

Example 2.14. To see the proposition in action, consider the following XNF clauses.

(a) Let $C_1 = \{x_1 + x_3,\ x_2 + x_3 + 1,\ x_1 + x_2\}$ and $C_2 = C_1 \setminus \{x_2 + x_3 + 1\}$ be two XNF clauses in $\mathbb{L}_3$. Because of

$$\begin{aligned} V_{C_1} &= \langle x_1 + x_3 + 1,\ x_2 + x_3,\ x_1 + x_2 + 1\rangle_{\mathbb{F}_2} \\ &= \langle x_1 + x_3 + 1,\ \ x_1 + x_2 + 1\rangle_{\mathbb{F}_2} \ = \ V_{C_2} \end{aligned}$$

we have $C_1 \equiv C_2$.

(b) For $C = \{x_1 + x_2 + x_4,\ x_1 + x_3,\ x_2 + x_4,\ x_3 + 1\} \subseteq \mathbb{L}_4$, we have

$$V_C \ = \ \langle x_1 + x_2 + x_4 + 1,\ x_1 + x_3 + 1,\ x_2 + x_4 + 1,\ x_3\rangle_{\mathbb{F}_2} \ = \ \langle 1,\ x_2 + x_4,\ x_1,\ x_3\rangle_{\mathbb{F}_2}$$

Hence $1 \in V_C$ implies $C \equiv \{0\}$.

(c) Let $C \subseteq \mathbb{L}_n$ be an XNF clause and $\ell_1, \ell_2 \in \mathbb{L}_n$. Then the clauses $C \cup \{\ell_1, \ell_2\} \equiv C \cup \{\ell_1, \ell_1 + \ell_2 + 1\}$ are equivalent, i.e., one can always *rewrite* the linerals of a clause via *addition* of another lineral from the clause.

In particular, part (a) of the example shows that an XNF clause may contain *superfluous* linerals. This motivates the following definition.

Definition 2.15. An XNF clause $C \subseteq \mathbb{L}_n$ is called **reduced** if $1 + C$ is a minimal generating system of I_C.

Remark 2.16. Let $C \subseteq \mathbb{L}_n$ be an XNF clause. If $1 \in V_C$, we have only the reduced clause $C = \{0\}$. For the case $1 \notin V_C$, Lemma 2.11 shows that C is reduced if and only if $1 + C$ is an $\mathbb{F}_2$-basis of V_C, i.e., if $\#C = \dim_{\mathbb{F}_2} V_C$. This shows that a reduced sub-clause $C' \subseteq C$ can be computed in polynomial time.

To conclude this section, let us count how many reduced *non-equivalent* XNF clauses exist in $\mathbb{L}_n$ and determine the size of the equivalence class of an XNF clause.

Remark 2.17.

(a) Let $C \subseteq \mathbb{L}_n$ be a reduced XNF clause with $d = \dim_{\mathbb{F}_2} V_C$. Denote the set of invertible $d \times d$ matrices over $\mathbb{F}_2$ by $\mathrm{GL}_d(\mathbb{F}_2)$. Then the number of distinct $\mathbb{F}_2$-bases of V_C is determined by $\frac{1}{d!}\#\,\mathrm{GL}_d(\mathbb{F}_2) = \frac{1}{d!}\prod_{i=0}^{d-1}(2^d - 2^i)$. In view of the previous remark, this is also the number of reduced clauses $C' \subseteq \mathbb{L}_n$ with $C' \equiv C$.

For instance, for $C = \{\ell_1, \ell_2\}$ with $d = \dim_{\mathbb{F}_2} V_C = 2$, there exist exactly three equivalent reduced clauses. These are $C \ \equiv \ \{\ell_1, \ell_1 + \ell_2 + 1\} \ \equiv \ \{\ell_2, \ell_1 + \ell_2 + 1\}$.

(b) Remember that $\dim_{\mathbb{F}_2} \mathbb{L}_n = n + 1$. Similar to (a), there are $\frac{1}{k!}\prod_{i=0}^{k-1}(2^{n+1} - 2^{i+1})$ linearly independent sets $S \subseteq \mathbb{L}_n$ with $1 \notin \langle S\rangle_{\mathbb{F}_2}$ and $\#S = k$. To get the total count of such subspaces $\langle S\rangle_{\mathbb{F}_2} \subseteq \mathbb{L}_n$, we divide this by the number of different bases of $\langle S\rangle_{\mathbb{F}_2}$, i.e., by $\frac{1}{k!}\#\,\mathrm{GL}_k(\mathbb{F}_2)$. Consequently, there are $\prod_{i=0}^{k-1}\frac{2^{n+1}-2^{i+1}}{2^k-2^i}$ distinct subspaces $U \subseteq \mathbb{L}_n$ with $1 \notin U$ and $\dim_{\mathbb{F}_2} U = k$.

Given Remarks 2.7 and 2.13, the number of non-equivalent reduced XNF clauses is

$$\begin{aligned} \#\left\{\prod C \mid C \subseteq \mathbb{L}_n\right\} &= 1 + \#\left\{U \subseteq \mathbb{L}_n \text{ subspace} \mid 1 \notin U\right\} \\ &= 1 + \sum_{k=0}^{n}\prod_{i=0}^{k-1}\tfrac{2^{n+1}-2^{i+1}}{2^k-2^i}. \end{aligned}$$

To conclude this section, note that the whole concept of equivalent clauses is superfluous in the setting of (non-tautological) CNF clauses. To be more precise, every non-tautological CNF clause $C \subseteq \mathbb{X}_n$ with $1 \notin C$ is reduced, and it is the unique CNF clause in this equivalence class, i.e., there are no two distinct reduced non-tautological equivalent CNF clauses.

3 The Proof System XLIN

In this paper we work with *propositional proof systems* as introduced in (Krajiček 2019) using the usual semantics of propositional logic. In particular, a proof system consists of

(1) a *syntax*, i.e., a set of formulae determining the language of the system,

(2) a set of *axioms*, i.e., a set of tautologies of the system, and

(3) a set of *rules of inference*, which determine how new formulae can be derived.

Here we allow only *sound* rules, i.e., ones where the new formula of the rule is semantically implied by its inputs.

For instance, the proof system RES, which forms the basis for conflict-driven SAT solvers, is defined as follows.

Definition 3.1. The proof system RES is defined by the following parts.

- The syntax is the set of CNF clauses, i.e., subsets of $\mathbb{X}_n$.
- The axioms are the *Boolean axioms* $\{x_i, x_i + 1\}$ for $i \in \{1, \dots, n\}$.
- The rules of inference consist only of the resolution rule res

$$\frac{F \cup \{\ell\} \quad G \cup \{\ell + 1\}}{F \cup G}\ (\text{res})$$

for $F, G \subseteq \mathbb{X}_n$ and $\ell \in \mathbb{X}_n$.

Before present XNF-based proof systems, let us recall the definition of a *proof* and of *polynomial simulation*.

Definition 3.2. Let ps be a proof system and let $F_1, \dots, F_m, G$ be formulae of ps.

(a) A tuple $\pi = (H_1, \dots, H_k)$ of formulae $H_1, \dots, H_k$ is called a ps-**proof** of G from so-called **initial premises** $F_1, \dots, F_m$, if $H_k = G$, and, for $i \in \{1, \dots, k\}$, the formula H_i is

- one of $F_1, \dots, F_m$,
- an axiom of ps, or
- the conclusion of a rule of inference of ps with premises $H_{i_1}, \dots, H_{i_s}$, where $1 \leq i_1, \dots, i_s < i$.

The formulae H_i are called the **lines** of the proof π.

If there is a ps-proof of G from $F_1, \dots, F_m$, we write $F_1, \dots, F_m \vdash_{\text{ps}} G$ or $\{F_1, \dots, F_m\} \vdash_{\text{ps}} G$.

(b) The **length** of a ps-proof $\pi = (H_1, \dots, H_k)$ is given by the number of lines, i.e., $\text{length}(\pi) = k$, and its **size** is measured as $\text{size}(\pi) = \sum_{i=1}^{k} \text{size}(H_i)$. Additionally, π is called **polynomial-size**, if $\text{size}(\pi)$ is polynomial in $\text{size}(F_1), \dots, \text{size}(F_m)$.

(c) A ps-proof of $\bot$ from $F_1, \dots, F_m$ is called a ps-**refutation** of $F_1, \dots, F_m$.

Definition 3.3. Let ps_1 and ps_2 be two proof systems.

(a) If every formula of ps_1 is a formula of ps_2 and there is a polynomial time algorithm translating any ps_1-proof π of a formula G from $F_1, \dots, F_m$ to a ps_2-proof π' of G from $F_1, \dots, F_m$, then we say ps_2 **p-simulates** ps_1.

(b) If ps_1 p-simulates ps_2, and ps_2 p-simulates ps_1, then we say ps_1 and ps_2 are **p-equivalent**.

Next, let us now focus on proof systems working with XNF clauses, and investigate their relations.
In (Horáček and Kreuzer 2018), the system SRES was introduced. It is based on a generalization of res.

Definition 3.4. The proof system SRES is defined by the following parts.

- The syntax is the set of XNF clauses, i.e., subsets of $\mathbb{L}_n$.

- The axioms are the Boolean axioms $\{x_i, x_i + 1\}$ for $i \in \{1, \ldots, n\}$.
- The rules of inference consist of s-resolution s-res for all $s \in \mathbb{N}_+$

$$\frac{F \cup \bigcup_{i=1}^{s} \{\ell_i\} \quad G \cup \bigcup_{i=1}^{s} \{\ell_i + 1\}}{F \cup G \cup \bigcup_{i=1}^{s-1} \{\ell_i + \ell_{i+1} + 1\}} \ (s\text{-res})$$

  with $F, G \subseteq \mathbb{L}_n$ and $\ell_1, \ldots, \ell_s \in \mathbb{L}_n$, and the weakening rule weak

$$\frac{H}{H \cup \{\ell\}} \ (\text{weak})$$

  for $H \subseteq \mathbb{L}_n$ and $\ell \in \mathbb{L}_n$.

Note that 1-res restricted to CNF clauses is the inference rule res.

Recall that a formula F *semantically implies* another formula G, written as $F \models G$, if $\mathcal{S}(F) \subseteq \mathcal{S}(G)$. The next proof system, introduced in (Itsykson and Sokolov 2014), translates RES in a straightforward way to XNF clauses and adds a *semantic* weakening rule.

Definition 3.5. The proof system RES($\oplus$) is defined by the following parts.

- The syntax is the set of XNF clauses, i.e., subsets of $\mathbb{L}_n$.
- The axioms are the Boolean axioms $\{x_i, x_i + 1\}$ for $i \in \{1, \ldots, n\}$.
- The rules of inference consist of the *semantic* weakening rule s-weak

$$\frac{F}{F'} \ (\text{s-weak})$$

  for $F, F' \subseteq \mathbb{L}_n$ with $F \models F'$, and the resolution rule res

$$\frac{F \cup \{\ell\} \quad G \cup \{\ell + 1\}}{F \cup G} \ (\text{res})$$

  for $F, G \subseteq \mathbb{L}_n$ and $\ell \in \mathbb{L}_n$.

The resolution rule here differs from the one in RES only in the syntax of the clauses it can be applied to. *Tree-like* and *regular* RES($\oplus$)-proofs are studied in a series of papers (Beame and Koroth 2023; Chattopadhyay et al. 2023; Efremenko et al. 2024; Garlík and Kołodziejczyk 2018; Gryaznov et al. 2024; Itsykson and Sokolov 2020; Khaniki 2022; Oparin 2016) with the goal to find classes of CNF instances with exponential lower bounds on RES($\oplus$)-refutations.

Finally, we introduce two new XNF-based proof systems, namely XRES and XLIN, as follows.

Definition 3.6. The proof system XRES is defined by the following parts.

- The syntax is the set of XNF clauses, i.e., subsets of $\mathbb{L}_n$.
- The axioms are the Boolean axioms $\{x_i, x_i + 1\}$ for $i \in \{1, \ldots, n\}$.
- The rules of inference consist of the weakening rule weak, the resolution rule res, and the rewriting rule rewr

$$\frac{F \cup \{\ell_1,\ \ell_2\}}{F \cup \{\ell_1,\ \ell_1 + \ell_2 + 1\}} \ (\text{rewr})$$

  for $F \subseteq \mathbb{L}_n$ and $\ell_1, \ell_2 \in \mathbb{L}_n$.

This system extends RES essentially by the rules weak and rewr. The latter is sound by Example 2.14.c.

Definition 3.7. The proof system XLIN is defined by the following parts.

- The syntax is the set of XNF clauses, i.e., subsets of $\mathbb{L}_n$.
- The axioms are $\{\ell, \ell + 1\}$ for $\ell \in \mathbb{L}_n$.
- The rules of inference consist of the weakening rule weak, the unit cancellation rule uc

$$\frac{F \cup \{1\}}{F}\ (\text{uc})$$

for $F \subseteq \mathbb{L}_n$, and the addition rule add

$$\frac{F \cup \{\ell_1\} \quad G \cup \{\ell_2\}}{F \cup G \cup \{\ell_1 + \ell_2\}}\ (\text{add})$$

for $F, G \subseteq \mathbb{L}_n$ and $\ell_1, \ell_2 \in \mathbb{L}_n$.

This system is different from RLIN, as introduced in (Horáček 2020), only in its set of axioms.
An overview of all proof systems of this section is given in Table 1.

Table 1. Overview of proof systems which admit XNF clauses.

proof system	axioms	rules of inference
RES($\oplus$)	$\{\ell, \ell + 1\}$ for $\ell \in \mathbb{X}_n$	s-weak, res
SRES	$\{\ell, \ell + 1\}$ for $\ell \in \mathbb{X}_n$	weak, sres
XRES	$\{\ell, \ell + 1\}$ for $\ell \in \mathbb{X}_n$	weak, rewr, res
XLIN	$\{\ell, \ell + 1\}$ for $\ell \in \mathbb{L}_n$	weak, uc, add

Before we show that all these proof systems are polynomially equivalent, consider the following example.

Example 3.8. Let $F = \{ C_1, C_2, C_3, C_4 \}$ be the XNF formula with clauses

$$C_1 = \{ x_1 + x_2 + 1,\ x_1 + x_3 \}, \quad C_2 = \{ x_2 + x_3 + 1 \}, \quad C_3 = \{ x_1,\ x_2 + 1 \}, \quad C_4 = \{ x_1 + 1,\ x_2 \}.$$

Then F is unsatisfiable, as the following proof using the sound inference rules rewr, res, and sres shows.

$$\dfrac{\dfrac{\dfrac{C_1}{\{x_1 + x_2 + 1, x_2 + x_3\}}\ (\text{rewr}) \qquad C_2}{\{x_1 + x_2 + 1\}}\ (\text{res}) \qquad \dfrac{C_3 \quad C_4}{\{x_1 + x_2\}}\ (\text{2-res})}{\emptyset}\ (\text{res})$$

The proof can also be re-written to use only *axioms* and *rules of inference* from one of the proof system RES($\oplus$), SRES, XRES, or XLIN. For instance, an XLIN-refutation is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{C_1 \quad \{x_1 + x_2 + 1, x_1 + x_2\}}{\{x_1 + x_2 + 1, x_2 + x_3\}}\ (\text{add}) \qquad C_2}{\{x_1 + x_2 + 1, 1\}}\ (\text{add})}{\{x_1 + x_2 + 1\}}\ (\text{uc}) \qquad \dfrac{\dfrac{\dfrac{C_3 \quad \{x_1, x_1 + 1\}}{\{x_1, x_1 + x_2\}}\ (\text{add}) \qquad \dfrac{C_4 \quad \{x_1 + 1, x_1\}}{\{x_1 + 1, x_1 + x_2\}}\ (\text{add})}{\{1, x_1 + x_2\}}\ (\text{add})}{\{x_1 + x_2\}}\ (\text{uc})}{\{1\}}\ (\text{add})}{\emptyset}\ (\text{uc})$$

To prove the polynomial equivalence of the proof systems, we start by providing a constant-size proof for each axiom $\{\ell, \ell + 1\}$ for $\ell \in \mathbb{L}_n$ using sres and weak from a given Boolean axiom.

Lemma 3.9. Let $\ell \in \mathbb{L}_n$. Then there is a constant-size proof

$$\{x_1, x_1 + 1\} \vdash_{\mathsf{weak,sres}} \{\ell, \ell + 1\}.$$

Proof. Let $\ell' = x_1 + \ell \in \mathbb{L}_n$. Then $\{\ell, \ell + 1\} = \{x_1 + \ell', x_1 + \ell' + 1\}$ has the following proof from $\{x_1, x_1 + 1\}$.

$$\dfrac{\dfrac{\{x_1, x_1 + 1\}}{\{x_1 + 1, \ell', x_1\}}\,(\mathsf{weak}) \qquad \dfrac{\{x_1, x_1 + 1\}}{\{x_1, \ell' + 1, x_1 + 1\}}\,(\mathsf{weak})}{\{x_1 + \ell', x_1 + \ell' + 1\}}\,(\mathsf{3\text{-}res})$$

□

Let us take a closer look at the inference rules weak, uc, add, res, sres, and rewr. To simplify the notation, for two sets of inference rules R_1 and R_2, we say that R_1 *p-simulates* R_2 if for every $\mathsf{ri} \in R_2$ there is a polynomial time algorithm that produces a proof of the conclusion of ri from its premises and the Boolean axioms using the inference rules in R_1. Similarly, R_1 and R_2 are called *p-equivalent*, if R_2 also p-simulates R_1.

Proposition 3.10. Under the Boolean axioms, we have the following p-simulations.

(a) The inference rules {weak, sres} p-simulate {rewr, add, uc}.

(b) The inference rules {add, uc} p-simulate {res}.

(c) The inference rules {rewr, res} p-simulate {sres}.

(d) The inference rules {weak, uc, rewr} are p-equivalent to {s-weak}.

Proof. Let $F, G \subseteq \mathbb{L}_n$ and $\ell, \ell_1, \ldots, \ell_s \in \mathbb{L}_n$. To show (a), note that add and uc are p-simulated by {weak, sres} due to (Horáček 2020, Proposition 5.25). Now Lemma 3.9 yields a constant-size proof of $\{x_1, x_1 + 1\} \vdash_{\mathsf{weak,sres}} \{\ell_1, \ell_1 + 1\}$. Additionally, we have

$$\dfrac{F \cup \{\ell_1, \ell_2\} \qquad \{\ell_1, \ell_1 + 1\}}{F \cup \{\ell_1\} \cup \{\ell_1 + \ell_2 + 1\}}\,(\mathsf{add})$$

which shows that rewr is also p-simulated by {weak, sres}.

For part (b), observe that res can be proven using uc and add as follows.

$$\dfrac{\dfrac{F \cup \{\ell_1\} \qquad G \cup \{\ell_1 + 1\}}{F \cup G \cup \{1\}}\,(\mathsf{add})}{F \cup G}\,(\mathsf{uc})$$

For (c), it suffices to consider the following proof of sres using $2s$ applications of the rule rewr and a single application of res.

$$\dfrac{\dfrac{\dfrac{F \cup \bigcup_{i=1}^{s} \{\ell_i\}}{F \cup \bigcup_{i=1}^{s-1} \{\ell_i\} \cup \{\ell_{s-1} + \ell_s + 1\}}\,(\mathsf{rewr})}{\vdots}\,(\mathsf{rewr})}{F \cup \{\ell_1\} \cup \bigcup_{i=1}^{s-1} \{\ell_i + \ell_{i+1} + 1\}}\,(\mathsf{rewr}) \qquad \dfrac{\dfrac{\dfrac{G \cup \bigcup_{i=1}^{s} \{\ell_i + 1\}}{G \cup \bigcup_{i=1}^{s-1} \{\ell_i + 1\} \cup \{\ell_{s-1} + \ell_s + 1\}}\,(\mathsf{rewr})}{\vdots}\,(\mathsf{rewr})}{G \cup \{\ell_1 + 1\} \cup \bigcup_{i=1}^{s-1} \{\ell_i + \ell_{i+1} + 1\}}\,(\mathsf{rewr})}{F \cup G \cup \bigcup_{i=1}^{s-1} \{\ell_i + \ell_{i+1} + 1\}}\,(\mathsf{res})$$

This proof can be constructed in polynomial time since s is bounded from above by the size of the input formulae. Finally, claim (d) follows from (Itsykson and Sokolov 2014, Remark 1) which shows that s-weak is p-simulated by weak, uc, and rewr. The converse p-simulation is clear. □

The proposition allows us to prove the promised p-equivalence.

Proposition 3.11. The propositional proof systems SRES, XRES, RES(⊕), and XLIN are p-equivalent.

Proof. First, we show that SRES, XRES and RES(⊕) are p-equivalent. By definition, these proof systems have the same Boolean axioms. Thus it suffices to show that their sets of inference rules are p-equivalent. By Proposition 3.10.c, we know that XRES p-simulates SRES. Proposition 3.10.a and 3.10.d show that SRES p-simulates RES(⊕). Lastly, observe that XRES is p-simulated by RES(⊕) due to Proposition 3.10.d. By transitivity of p-simulation, all three proof systems are p-equivalent.

For an XNF clause $C \subseteq \mathbb{L}_n$ and $\ell_1, \ell_2 \in \mathbb{L}_n$, we have

$$C \cup \{\ell_1, \ell_2\}, \{\ell_1, \ell_1 + 1\} \vdash_{\mathsf{add}} C \cup \{\ell_1, \ell_1 + \ell_2 + 1\}.$$

Thus the rule of inference rewr is p-simulated by XLIN. In combination with Proposition 3.10.b, it follows that XLIN p-simulates XRES. Finally, Lemma 3.9 and Proposition 3.10.a show that XLIN is p-simulated by SRES. The p-equivalence of XRES and SRES implies that XLIN is also p-equivalent to SRES, XRES, and RES(⊕). □

Remark 3.12.

(a) By (Horáček 2020, Proposition 3.30), SRES is *implicationally* and *refutationally complete*, i.e., for formulae $F_1, \ldots, F_m, G$ we have $F_1, \ldots, F_m \models G$ if and only if $F_1, \ldots, F_m \vdash_{\mathsf{SRES}} G$. Thus, the above p-equivalence also implies that XRES, RES(⊕), and XLIN are implicationally and refutationally complete.

(b) The proposition implies that exponential lower bounds on the size of RES(⊕)-refutations are also exponential lower bounds for SRES-, XRES-, and XLIN-refutations. To the best of our knowledge, such lower bounds have only been found for *tree-like* refutations, see (Beame and Koroth 2023; Chattopadhyay et al. 2023; Efremenko et al. 2024; Garlík and Kołodziejczyk 2018; Gryaznov et al. 2024; Itsykson and Sokolov 2020; Khaniki 2022; Oparin 2016). Most notable, in (Itsykson and Sokolov 2020) an exponential gap in the size of the shortest refutation between RES(⊕) and RES was found for a class of CNF formulae encoding the existence of perfect matchings in graphs with an odd number of vertices. Thus, SRES, XRES, and XLIN are *exponentially* stronger than RES, and considered *fairly strong* proof systems.

(c) Recall that, for a proof system ps, lower bounds on ps-refutation sizes correspond to lower bounds on the running time of any SAT solver which reasons within ps. Therefore the above suggests that an XLIN-based solver might have an exponential advantage over traditional CNF-based solving which is based on RES.

4 Conflict-Driven XNF SAT Solving

In order to develop conflict-driven XNF SAT solvers, we use a stronger notion of assignments compared to the one used in CNF-based solving. Section 4.1 introduces *lineral assignments* and discusses unit and reason XNF clauses. In this terminology we obtain a concise description of *Gaußian Constraint Propagation* from (Horáček 2020) and formalize its execution using *lineral trails* in Section 4.2. These are used in Section 4.3 as the basis for a XNF-based conflict-learning algorithm which returns an *asserting* XNF clause. Altogether, we combine these methods to get a conflict-driven XNF SAT solving framework, called CDXCL, in Section 4.4.

4.1 Assignments and Reason Clause Decompositions

Instead of assignments which map from the set of variables to the truth values true and false, or equivalently to 0 and 1 in $\mathbb{F}_2$, we use the following notion of *lineral assignments*, assigning variables to linerals. Recall that a set of linerals $S \subseteq \mathbb{L}_n$ is called LT_σ-interreduced if $\mathrm{LT}_\sigma(f) \neq \mathrm{LT}_\sigma(g)$ for all distinct $f, g \in S$. Moreover, the *normal remainder* of f with respect to a tuple $(g_1, \ldots, g_k) \in \mathbb{B}_n^k$ is denoted by $\mathrm{NR}_\sigma(f, (g_1, \ldots, g_k))$ (see (Andraschko et al. 2024; Kreuzer and Robbiano 2000)). In our setting, where all polynomials are linear, the map $f \mapsto \mathrm{NR}_\sigma(f, (g_1, \ldots, g_k))$ is a *projection* from $\mathbb{L}_n = \langle x_1, \ldots, x_n, 1\rangle_{\mathbb{F}_2}$ to $\langle \{x_1, \ldots, x_n, 1\} \setminus \{\mathrm{LT}_\sigma(g_1), \ldots, \mathrm{LT}_\sigma(g_k)\}\rangle_{\mathbb{F}_2}$.

Definition 4.1.

(a) An LT_σ-interreduced set $L \subseteq \mathbb{L}_n$ is called a **(lineral) σ-assignment**. If $1 \in L$, we say L is an **invalid** σ-assignment, otherwise it is a **valid** σ-assignment.

If σ is not relevant in the context, we simply say that L is an assignment.

(b) Let $L \subseteq \mathbb{L}_n$ be a valid assignment. If $\dim_{\mathbb{F}_2}\langle L\rangle_{\mathbb{F}_2} = n$, then L is called a **complete** assignment, otherwise a **partial** assignment.

(c) Let $L \subseteq \mathbb{L}_n$ be a σ-assignment. The map $\alpha_L \colon \mathbb{L}_n \to \mathbb{L}_n$ with $\ell \longmapsto \mathrm{NR}_\sigma(\ell, L)$ is called the **assignment map of** L.

(d) Let $\ell \in \mathbb{L}_n$ be a lineral, $C \subseteq \mathbb{L}_n$ be an XNF clause, $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be an XNF formula, and $L \subseteq \mathbb{L}_n$ be an assignment. Then the lineral $\ell|_L = \alpha_L(\ell)$ is called ℓ **under** L, the XNF clause $C|_L = \alpha_L(C)$ is called C **under** L, and the XNF formula $F|_L = \{C'|_L \mid C' \in F\}$ is called F **under** L.

(e) For a set $L \subseteq \mathbb{L}_n$, the XNF formula $\widehat{L} = \big\{\{\ell\} \mid \ell \in L\big\}$ is called the L**-formula**. It is a logical representation of $\langle L\rangle$, i.e., we have $\mathcal{Z}(L) = \mathcal{S}\big(\widehat{L}\,\big)$.

(f) An assignment $L \subseteq \mathbb{X}_n$ is called a **variable assignment**.

Note that traditional CNF-based SAT solvers work exclusively with variable assignments $A \subseteq \mathbb{X}_n$.

Remark 4.2. Let $L \subseteq \mathbb{L}_n$ be a σ-assignment.

(a) Since L is LT_σ-interreduced and consists only of linear polynomials, it forms a σ-Gröbner basis of $\langle L\rangle$. Hence, the normal remainder of $\ell \in \mathbb{L}_n$ with respect to L is well-defined even without a specific order on the elements of L. Therefore, the assignment map of L in Definition 4.1.c is well-defined.

(b) Additionally, we have $\mathrm{NR}_\sigma(1, L) = 0$, if $1 \in \langle L\rangle_{\mathbb{F}_2}$. As $\mathrm{LT}_\sigma(\ell) >_\sigma 1$ for all non-constant $\ell \in \mathbb{L}_n$, this implies $1 \in L$. Altogether, we have

$$\alpha_L(1) = 0 \quad\Longleftrightarrow\quad 1 \in \langle L\rangle_{\mathbb{F}_2} \quad\Longleftrightarrow\quad 1 \in L \quad\Longleftrightarrow\quad L \text{ is invalid.}$$

Next we take a look at properties of the assignment map.

Remark 4.3. Let $L \subseteq \mathbb{L}_n$ be a valid σ-assignment. Then the assignment map α_L is an $\mathbb{F}_2$-linear projection. Its kernel is $\ker(\alpha_L) = \langle L\rangle_{\mathbb{F}_2}$ and the image is $\mathrm{im}(\alpha_L) = \langle \{x_1, \ldots, x_n, 1\} \setminus \mathrm{LT}_\sigma(L)\rangle_{\mathbb{F}_2}$. Therefore $\mathbb{L}_n = \ker(\alpha_L) \oplus \mathrm{im}(\alpha_L)$ and the pre-image of a lineral $\ell \in \mathrm{im}(\alpha_L)$ is given by $\alpha_L^{-1}(\ell) = \ell + \langle L\rangle_{\mathbb{F}_2}$, i.e., we have $\ell|_L \in \ell + \langle L\rangle_{\mathbb{F}_2}$.

Example 4.4. The set $L = \big\{x_1 + x_2 + x_4,\, x_2 + x_3 + x_4,\, x_4 + x_5,\, x_6\big\} \subseteq \mathbb{L}_6$ is a valid Lex-assignment with $\mathrm{LT}_{\mathsf{Lex}}(L) = \big\{x_1,\, x_2,\, x_4,\, x_6\big\}$. The assignment map α_L of L is $\mathbb{F}_2$-linear and uniquely determined by

$$\begin{aligned} \alpha_L(1) &= 1, & \alpha_L(x_1) &= x_3, & \alpha_L(x_2) &= x_3 + x_5, & \alpha_L(x_3) &= x_3, \\ & & \alpha_L(x_4) &= x_5, & \alpha_L(x_5) &= x_5, & \alpha_L(x_6) &= 0. \end{aligned}$$

For the XNF clause $C = \big\{x_1 + x_2 + x_4 + 1,\, x_6 + 1\big\} \subseteq \mathbb{L}_6$, we have

$$C|_L = \alpha_L(C) = \big\{\alpha_L(x_1 + x_2 + x_4 + 1),\, \alpha_L(x_6 + 1)\big\} = \{1\} \equiv \bot.$$

Let us point out that the definition of $C|_L$ for an XNF clause C and an assignment L is *sound* in the following sense.

Remark 4.5. Let $L \subseteq \mathbb{L}_n$ be a valid σ-assignment and $C \subseteq \mathbb{L}_n$ be a clause.

(a) Let $\ell \in C$. Then $\ell|_L$ arises from ℓ via addition of linerals from L. This shows $\widehat{L} \cup \{C\} \vdash_{\mathsf{add}} C|_L$ and

$$\widehat{L} \cup \{C\} \;\models\; C|_L.$$

In particular, this can be backed with a polynomial-size XLIN-proof, since every lineral $\ell \in C$ is *reduced* by each lineral of L at most once.

(b) Conversely, for every $\ell \in C$ one can derive ℓ from $\ell|_L$ via addition of linerals from L. Using analogous arguments, we get

$$\widehat{L} \cup \{C|_L\} \;\models\; C.$$

A polynomial-size XLIN-proof for this fact can be derived similarly to (a).

(c) For all propositional logic formulae F, G, and H with $F \wedge G \models H$, we have $F \models G \to H$. Therefore parts (a) and (b) imply

$$\widehat{L} \;\models\; C \leftrightarrow C|_L,$$

i.e., for all $a \in \mathcal{S}(\widehat{L}) = \mathcal{Z}(L)$ it follows that a satisfies C if and only if a satisfies $C|_L$.

If L is an invalid assignment the same holds true, since $1 \in L$ implies $\widehat{L} \models \{1\} \equiv \bot$, and thus $\widehat{L}$ semantically implies every other formula. In particular this means $\widehat{L} \models C|_L \leftrightarrow C$.

Next we generalize the concept of *unit* and *conflict clauses* to XNF clauses and lineral assignments.

Definition 4.6. Let $L \subseteq \mathbb{L}_n$ be an assignment and $C \subseteq \mathbb{L}_n$ be an XNF clause.

(a) If $C \equiv \{\ell\}$ for some $\ell \in \mathbb{L}_n$, then C is called a **unit clause**. If $C \equiv \top$, then C is called the **trivial** unit clause.

(b) If $C|_L \equiv \{\ell\}$ for some $\ell \in \mathbb{L}_n$ is a unit clause, then C is called a **reason clause for** ℓ **under** L, and ℓ is the **implied lineral of** C **under** L.

(c) If $C|_L \equiv \bot$ is unsatisfiable, then C is called a **conflict clause under** L and **violated by** L.

(d) If $C|_L \equiv \top$ is the trivial unit clause, then C is said to be **satisfied by** L.

Example 4.7. Let $C = \{x_1 + x_2,\ x_2 + x_3,\ x_5 + 1\} \subseteq \mathbb{L}_5$ be an XNF clause and let $L = \{x_1 + x_4,\ x_3 + x_4,\ x_5\}$ be a Lex-assignment. Then C is a reason clause for $x_2 + x_4$ under L, as

$$C|_L \;=\; \big\{(x_1 + x_2)|_L,\ (x_2 + x_3)|_L,\ (x_5 + 1)|_L\big\} \;=\; \{x_2 + x_4,\ 1\} \;\equiv\; \{x_2 + x_4\}.$$

In the CNF setting a reason clause can be uniquely decomposed into two parts: the literals which are violated by the assignment, and the implied literal. There is also a similar useful decomposition for XNF reason clauses. The only difference is that instead of the clause C itself, we decompose its associated subspace $V_C = \langle 1 + C\rangle_{\mathbb{F}_2}$, i.e., the decomposition is invariant under the equivalence class of C, see Proposition 2.12.

Proposition 4.8. Let L be a valid σ-assignment and $C \subseteq \mathbb{L}_n$ be an XNF clause.

(a) Let $\ell \in \mathbb{L}_n \setminus \{0\}$ be a lineral. Then C is a reason clause for ℓ under L if and only if there is $\ell_C \in \mathbb{L}_n$ with $\ell_C|_L = \ell$ and $V_C \;=\; \big(V_C \cap \langle L\rangle_{\mathbb{F}_2}\big) \,\oplus\, \langle \ell_C + 1\rangle_{\mathbb{F}_2}$.

(b) The clause C is a conflict clause under L if and only if $V_C \subseteq \langle L\rangle_{\mathbb{F}_2}$.

Proof. If C is a reason clause for ℓ under L, then $C|_L \equiv \{\ell\}$. From Proposition 2.12 we get $V_{C|_L} = V_{\{\ell\}}$ or $1 \in V_{C|_L} \cap V_{\{\ell\}}$. The latter implies $1 \in V_{\{\ell\}} = \langle \ell + 1\rangle_{\mathbb{F}_2}$, i.e., $\ell = 0$. This contradicts $\ell \in \mathbb{L}_n \setminus \{0\}$, and we have

$$\alpha_L(V_C) = \langle \alpha_L(1 + C)\rangle_{\mathbb{F}_2} = \langle 1 + \alpha_L(C)\rangle_{\mathbb{F}_2} = V_{\alpha_L(C)} = V_{C|_L} = V_{\{\ell\}} = \langle \ell + 1\rangle_{\mathbb{F}_2}.$$

Hence the restriction $\tilde{\alpha}_L = \alpha_L|_{V_C}$ of α_L to V_C defines an epimorphism $V_C \to \langle \ell+1\rangle_{\mathbb{F}_2}$ with $\ker(\tilde{\alpha}_L) = V_C \cap \ker(\alpha_L) = V_C \cap \langle L\rangle_{\mathbb{F}_2}$. By the rank-nullity theorem an $\mathbb{F}_2$-basis B of $V_C \cap \langle L\rangle_{\mathbb{F}_2}$ can be extended with a lineral $\ell_C + 1 \in V_C$ to a basis of V_C. This yields $V_C = \langle B\rangle_{\mathbb{F}_2} \oplus \langle \ell_C + 1\rangle_{\mathbb{F}_2} = \big(V_C \cap \langle L\rangle_{\mathbb{F}_2}\big) \oplus \langle \ell_C + 1\rangle_{\mathbb{F}_2}$ with $\ell_C|_L = \ell$ because of

$$\langle \ell + 1\rangle_{\mathbb{F}_2} = \alpha_L(V_C) = \langle \alpha_L(B)\rangle_{\mathbb{F}_2} + \langle \alpha_L(\ell_C + 1)\rangle_{\mathbb{F}_2} = \langle \alpha_L(\ell_C) + 1\rangle_{\mathbb{F}_2}.$$

Conversely, let $\ell_C \in \mathbb{L}_n$ with $\ell_C|_L = \ell$ and $V_C = \big(V_C \cap \langle L\rangle_{\mathbb{F}_2}\big) \oplus \langle \ell_C + 1\rangle_{\mathbb{F}_2}$. With Proposition 2.12 and

$$V_{C|_L} = \alpha_L(V_C) = \alpha_L(V_C \cap \langle L\rangle_{\mathbb{F}_2}) + \alpha_L(\langle \ell_C + 1\rangle_{\mathbb{F}_2}) = \langle 0\rangle_{\mathbb{F}_2} + \langle \ell + 1\rangle_{\mathbb{F}_2} = V_{\{\ell\}}$$

we get $C|_L \equiv \{\ell\}$. Thus, C is a reason clause for ℓ under L. This finishes the proof of (a).

Next we show (b). By definition $C \subseteq \mathbb{L}_n$ is a conflict clause under L if it is a reason clause for 1 under L. As shown in part (a) there is $\ell_C \in \mathbb{L}_n$ with $V_C = \big(V_C \cap \langle L\rangle_{\mathbb{F}_2}\big) \oplus \langle \ell_C + 1\rangle_{\mathbb{F}_2}$ and $\ell_C|_L = 1$. The latter implies $\ell_C \in 1 + \langle L\rangle_{\mathbb{F}_2}$, i.e., $\ell_C + 1 \in \langle L\rangle_{\mathbb{F}_2}$. Thus $\ell_C + 1 \in V_C \cap \langle L\rangle_{\mathbb{F}_2}$ and we get $V_C = V_C \cap \langle L\rangle_{\mathbb{F}_2} \subseteq \langle L\rangle_{\mathbb{F}_2}$. On the contrary, $V_C \subseteq \langle L\rangle_{\mathbb{F}_2}$ implies $C \subseteq 1 + \langle L\rangle_{\mathbb{F}_2}$, i.e., $C|_L = \{1\}$. Hence $C|_L \equiv \bot$ is unsatisfiable. □

The case where $C \subseteq \mathbb{L}_n$ is a reason clause for $0 \in \mathbb{L}_n$ under an assignment $L \subseteq \mathbb{L}_n$ is not of interest for us, since this amounts to C being satisfied by L.

Remark 4.9. Let $L \subseteq \mathbb{L}_n$ be a valid assignment and $C \subseteq \mathbb{L}_n$ be an XNF clause.

(a) The clause C is reason clause for some $\ell \in \mathbb{L}_n \setminus \mathbb{F}_2$ under L if and only if $\dim_{\mathbb{F}_2} V_C/\big(V_C \cap \langle L\rangle_{\mathbb{F}_2}\big) = 1$.

(b) The clause C is a conflict clause under L if and only if $\dim_{\mathbb{F}_2} V_C/\big(V_C \cap \langle L\rangle_{\mathbb{F}_2}\big) = 0$.

In particular, $C|_L$ is a non-trivial unit clause if and only if $\dim_{\mathbb{F}_2} V_C/\big(V_C \cap \langle L\rangle_{\mathbb{F}_2}\big) \leq 1$.

In view of the characterization of Proposition 4.8, we introduce the following notion.

Definition 4.10. Let $C \subseteq \mathbb{L}_n$ be a reason clause for a lineral $\ell \in \mathbb{L}_n \setminus \{0\}$ under a valid assignment $L \subseteq \mathbb{L}_n$. By Proposition 4.8.a, there is some $\ell_C \in \mathbb{L}_n$ with

$$V_C = \big(V_C \cap \langle L\rangle_{\mathbb{F}_2}\big) \oplus \langle \ell_C + 1\rangle_{\mathbb{F}_2} \quad \text{and} \quad \ell_C|_L = \ell.$$

This direct sum representation of V_C is called a **reason clause decomposition of V_C (with respect to L)**, and the lineral ℓ_C is called a **reason lineral of C under L**.

As we show next, reason clause decompositions and reason linerals are not uniquely determined.

Remark 4.11. Let $C \subseteq \mathbb{L}_n$ be a reason clause for a lineral $\ell \in \mathbb{L}_n \setminus \{0\}$ under a valid assignment $L \subseteq \mathbb{L}_n$.

(a) Let ℓ_C be a reason lineral of C under L. Then the decomposition $V_C = \big(V_C \cap \langle L\rangle_{\mathbb{F}_2}\big) \oplus \langle \ell_C + 1\rangle_{\mathbb{F}_2}$ implies

$$V_C \,/\, \big(V_C \cap \langle L\rangle_{\mathbb{F}_2}\big) = \langle\, \overline{\ell_C + 1}\,\rangle_{\mathbb{F}_2}.$$

(b) For reason linerals ℓ_C, ℓ'_C of C under L this yields $\overline{\ell_C} = \overline{\ell'_C}$ in $V_C \,/\, \big(V_C \cap \langle L\rangle_{\mathbb{F}_2}\big)$. Consequently, reason linerals are unique *modulo* $V_C \cap \langle L\rangle_{\mathbb{F}_2}$.

Let us apply these propositions and remarks to the reason clause from Example 4.7.

Example 4.12. The XNF clause $C = \{x_1 + x_2,\ x_2 + x_3,\ x_5 + 1\} \subseteq \mathbb{L}_5$ with associated subspace $V_C = \langle x_1 + x_2 + 1,\ x_2 + x_3 + 1,\ x_5\rangle_{\mathbb{F}_2}$ is a reason clause for $x_2 + x_4$ under the Lex-assignment $L = \{x_1 + x_4,\ x_3 + x_4,\ x_5\}$. We have $V_C \cap \langle L\rangle_{\mathbb{F}_2} = \langle x_1 + x_3,\ x_5\rangle_{\mathbb{F}_2}$ with reason linerals $x_1 + x_2$ and $x_2 + x_3$ of C under L. This yields the two reason clause decompositions

$$\begin{aligned} V_C &= \langle x_1 + x_3,\ x_5\rangle_{\mathbb{F}_2} \oplus \langle (x_1 + x_2) + 1\rangle_{\mathbb{F}_2} \\ &= \langle x_1 + x_3,\ x_5\rangle_{\mathbb{F}_2} \oplus \langle (x_2 + x_3) + 1\rangle_{\mathbb{F}_2}. \end{aligned}$$

In particular, we have $(x_1 + x_2) + (x_2 + x_3) \in V_C \cap \langle L\rangle_{\mathbb{F}_2}$, i.e., $\overline{x_1 + x_2} = \overline{x_2 + x_3}$.

To conclude this subsection, observe that reason clause decompositions of V_C give rise to equivalent clauses of C which contain exactly one reason lineral.

Remark 4.13. Let $C \subseteq \mathbb{L}_n$ be a reason clause for a lineral $\ell \in \mathbb{L}_n \setminus \mathbb{F}_2$ under a valid assignment $L \subseteq \mathbb{L}_n$ with a reason lineral $\ell_C \in \mathbb{L}_n$. Then we have a reason clause decomposition $V_C = \big(V_C \cap \langle L\rangle_{\mathbb{F}_2}\big) \oplus \langle \ell_C + 1\rangle_{\mathbb{F}_2}$. This shows that for every basis B of $V_C \cap \langle L\rangle_{\mathbb{F}_2}$ the XNF clause $C' = (1 + B) \cup \{\ell_C\}$ satisfies $C' \equiv C$. Then $\ell_C \in C'$ is the unique lineral of C' which is also a reason lineral of C. Moreover, $C \equiv C'$ shows $C \vdash_{\text{s-weak}} C'$, and Proposition 3.11 implies the existence of a polynomial-size proof $C \vdash_{\mathsf{XLIN}} C'$.

4.2 Gaußian Constraint Propagation

To *propagate* lineral assignments to XNF clauses, we use an analogue of *Boolean Constraint Propagation* (BCP) called *Gaußian Constraint Propagation* (GCP). It was originally proposed in (Horáček 2020, Algorithm 5.7), and is translated to our terminology in Algorithm 1. Its finiteness is clear, and the correctness follows directly from the next remark.

Remark 4.14. Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be an XNF formula and $L \subseteq \mathbb{L}_n$ be a σ-assignment.

(a) For every $\ell \in \mathbb{L}_n$ we have $\mathrm{LT}_\sigma(\ell|_L) \notin \mathrm{LT}_\sigma(L)$, i.e., the set $L \cup \{\ell|_L\}$ is LT_σ-interreduced and therefore also a σ-assignment.

(b) By Remark 4.5 and the definition of the L-formula $\widehat{L} = \big\{\{\ell\} \mid \ell \in L\big\}$ we have

$$\mathcal{S}(F) \cap \mathcal{Z}(L) \;=\; \mathcal{S}\big(F \cup \widehat{L}\big) \;=\; \mathcal{S}\big(F|_L \cup \widehat{L}\big) \;=\; \mathcal{S}(F|_L) \cap \mathcal{Z}(L).$$

Algorithm 1: GCP – Gaußian Constraint Propagation

Input : An XNF formula $F \subseteq \mathcal{P}(\mathbb{L}_n)$, a σ-assignment $L \subseteq \mathbb{L}_n$.
Output: A σ-assignment L' with $L \subseteq L'$ and $\mathcal{S}(F) \cap \mathcal{Z}(L) = \mathcal{S}(F) \cap \mathcal{Z}(L')$.

1 **while** *there is a clause* $C \in F$ *s.t.* $C|_L \equiv \{\ell\}$ *is a non-trivial unit clause* **do**
2 | Adjoin ℓ to L.
3 **return** L.

Remark 4.15. From the very definition of GCP, we see that it generalizes *Boolean Constraint Propagation* (BCP), independently of the chosen term ordering σ.

Let us now highlight that neither the choice of the term ordering nor the order in which clauses are chosen in line 1 affects the *deductive power* of GCP_σ. This is also why we do not list the term ordering explicitly as an input and write GCP instead of GCP_σ.

Remark 4.16. Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be an XNF formula and L be a valid assignment.

(a) Let σ and τ be two term orderings. Let L_σ and L_τ be valid σ- and τ-assignments with $\langle L_\sigma\rangle_{\mathbb{F}_2} = \langle L\rangle_{\mathbb{F}_2} = \langle L_\tau\rangle_{\mathbb{F}_2}$. By Remark 4.9 we have for every XNF clause $C \in F$:

$$\begin{aligned} C|_{L_\sigma} \text{ is a non-trivial unit clause} \quad &\Longleftrightarrow \quad \dim_{\mathbb{F}_2} V_C/\big(V_C \cap \langle L_\sigma\rangle_{\mathbb{F}_2}\big) \le 1 \\ &\Longleftrightarrow \quad \dim_{\mathbb{F}_2} V_C/\big(V_C \cap \langle L_\tau\rangle_{\mathbb{F}_2}\big) \le 1 \\ &\Longleftrightarrow \quad C|_{L_\tau} \text{ is a non-trivial unit clause.} \end{aligned}$$

Now let $C \in F$ be a reason clause for $\ell \in \mathbb{L}_n \setminus \{0\}$ under L_σ. Then C is also a reason clause under L_τ, and there is an implied lineral $\ell' \in \mathbb{L}_n$ which can be different from ℓ. A reason clause decomposition

of V_C with respect to L_σ is also a reason clause decomposition with respect to L_τ, as $\langle L_\sigma \rangle_{\mathbb{F}_2} = \langle L_\tau \rangle_{\mathbb{F}_2}$. Thus Remark 4.11 shows that the corresponding reason linerals must be equal *modulo* $V_C \cap \langle L \rangle_{\mathbb{F}_2} \subseteq \langle L \rangle_{\mathbb{F}_2}$. This yields $\langle L \cup \{\ell\} \rangle_{\mathbb{F}_2} = \langle L \cup \{\ell'\} \rangle_{\mathbb{F}_2}$. Applying this argument iteratively to all newly adjoined linerals of line 2, we get

$$\langle \mathrm{GCP}_\sigma(F, L_\sigma) \rangle_{\mathbb{F}_2} = \langle \mathrm{GCP}_\tau(F, L_\tau) \rangle_{\mathbb{F}_2},$$

i.e., independent of the term ordering GCP derives the same subspace of linerals.

(b) The order in which unit clauses in line 1 are chosen affects the implied linerals and, thereby, the output of GCP. Consider an iteration of GCP, where two clauses $C_1, C_2 \in F$ satisfy $C_1|_L \equiv \{\ell_1\}$ and $C_2|_L \equiv \{\ell_2\}$. If we first pick C_1 and then C_2, we adjoin ℓ_1 and $\ell_2|_{\ell_1}$ to get the assignment $L \cup \{\ell_1, \ell_2|_{\ell_1}\}$. Observe that $\ell_2|_{\ell_1} \in \{\ell_1 + \ell_2, \ell_2\}$ implies $\langle L \cup \{\ell_1, \ell_2|_{\ell_1}\} \rangle_{\mathbb{F}_2} = \langle L \rangle_{\mathbb{F}_2} + \langle \ell_1, \ell_2 \rangle_{\mathbb{F}_2}$. Processing the clauses the other way around, we get the assignment $L \cup \{\ell_2, \ell_1|_{\ell_2}\}$. Analogous to above we have $\langle L \cup \{\ell_2, \ell_1|_{\ell_2}\} \rangle_{\mathbb{F}_2} = \langle L \rangle_{\mathbb{F}_2} + \langle \ell_1, \ell_2 \rangle_{\mathbb{F}_2}$.

Hence, after both clauses have been processed the same subspace $\langle L \rangle_{\mathbb{F}_2} + \langle \ell_1, \ell_2 \rangle_{\mathbb{F}_2}$ is derived. Consequently, the subspace of linerals derived by GCP is invariant under *transpositions* on the order in which non-trivial unit clauses are processed. Since transpositions generate all permutations, the subspace $\langle \mathrm{GCP}_\sigma(F, L) \rangle_{\mathbb{F}_2}$ does not depend on the order in which clauses are processed in line 1.

Altogether, the subspace $\langle \mathrm{GCP}_\sigma(F, L) \rangle_{\mathbb{F}_2}$ is determined uniquely by F and $\langle L \rangle_{\mathbb{F}_2}$.

Example 4.17. Let F be the XNF formula in $\mathbb{L}_8$ with clauses

$$\begin{aligned} C_1 &= \{x_5 + 1,\ x_3 + x_8\}, & C_2 &= \{x_3 + 1,\ x_5 + 1\}, \\ C_3 &= \{x_3,\ x_1 + x_4\}, & C_4 &= \{x_3 + x_5,\ x_2 + x_4 + x_7\}, \\ C_5 &= \{x_1 + x_3 + x_5 + x_6,\ x_1 + x_3 + x_4\}, & C_6 &= \{x_1 + x_5 + x_6,\ x_2 + x_4 + x_7 + 1\}. \end{aligned}$$

Under the Lex-assignment $L = \{x_1 + x_2,\ x_5\}$ we get an invalid assignment

$$\mathrm{GCP}_{\mathsf{Lex}}(F, L) = L \cup \{\, x_3 + x_8,\ x_8 + 1,\ x_2 + x_4,\ x_7,\ x_4 + x_5 + x_6 + 1,\ 1 \,\}.$$

The order in which we processed the clauses is schematically illustrated in Figure 1. Note that $1 \in \mathrm{GCP}_{\mathsf{Lex}}(F, L)$ shows that $F|_L$ is unsatisfiable.

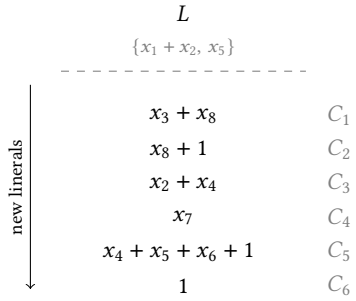


Fig. 1. Execution of GCP with the inputs given in Example 4.17.

Now we have all the ingredients to present a DPLL-based algorithm for solving XNF formulae. As in the CNF setting, the algorithm explores the search space by taking and propagating *decisions*. Here, we consider *lineral* decisions of the following kind.

Definition 4.18. Let $L \subseteq \mathbb{L}_n$ be an assignment. A lineral $\ell \in \mathbb{L}_n$ with $\ell|_L \notin \mathbb{F}_2$, i.e., $\ell \notin \langle L \rangle_{\mathbb{F}_2}$ and $\ell + 1 \notin \langle L \rangle_{\mathbb{F}_2}$, is called a **decision** for L.

Remark 4.19. For every assignment $L \subseteq \mathbb{L}_n$ and every decision $\ell \in \mathbb{L}_n$ for L, we know that $\mathbb{F}_2^n = \mathcal{Z}(\ell) \dot\cup \mathcal{Z}(\ell+1)$ is a disjoint union. Therefore, we get

$$\mathcal{S}(F) \cap \mathcal{Z}(L) \;=\; \big(\mathcal{S}(F) \cap \mathcal{Z}(L \cup \{\ell\})\big) \;\dot\cup\; \big(\mathcal{S}(F) \cap \mathcal{Z}(L \cup \{\ell+1\})\big).$$

Thus we can explore $\mathcal{S}(F)$ by recursively splitting this set using decisions as above. We use GCP to propagate the decisions and derive new linerals to speed up the process. In particular, $L \cup \{\ell\}$ and $L \cup \{\ell+1\}$ both contain more $\mathbb{F}_2$-linear independent elements, i.e., we can only branch finitely often, and we eventually gain a complete algorithm. This rather straightforward DPLL-based approach to solve XNF SAT is summarized in Algorithm 2. This construction was considered before in (Horáček 2020, Algorithm 5.8); however, it allowed only a weaker definition of a decision.

Algorithm 2: XDPLL– DPLL-based XNF SAT Solving

Input : An XNF formula $F \subseteq \mathcal{P}(\mathbb{L}_n)$, a σ-assignment $L \subseteq \mathbb{L}_n$.
Output: UNSAT or $a \in \mathcal{S}(F) \cap \mathcal{Z}(L)$.

1 Compute $L' = \mathsf{GCP}(F, L)$. `propagation`
2 **if** L' *is invalid* **then return** UNSAT.
3 **if** L' *is a complete assignment* **then return** $a \in \mathcal{Z}(L')$.
4 Choose a decision $\ell \in \mathbb{L}_n$ for L' and let $\ell' = \ell|_{L'}$. `decision`
5 **if** $\mathsf{XDPLL}(F, L' \cup \{\ell'\})$ *returns* $a \in \mathbb{F}_2^n$ **then return** a.
6 **else return** $\mathsf{XDPLL}(F, L' \cup \{\ell' + 1\})$.

Note that a few recursive calls can be avoided by replacing the condition in line 3 with L' *satisfies* F.

Proposition 4.20. Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be an XNF formula, and $L \subseteq \mathbb{L}_n$ be an assignment. Then XDPLL is an algorithm. It returns $\mathsf{UNSAT} = \mathsf{XDPLL}(F, L)$ if and only if $F|_L$ is unsatisfiable or L is invalid. Otherwise, it returns a satisfying assignment $a \in \mathcal{S}(F) \cap \mathcal{Z}(L)$.

The proof is straightforward. It is also a consequence of Proposition 4.41, see Remark 4.42.c.

In the following we show that an execution of XDPLL with an unsatisfiable formula $F \subseteq \mathcal{P}(\mathbb{L}_n)$ as input gives rise to an XLIN-refutation of F whose size is polynomial in the running time of $\mathsf{XDPLL}(F)$.

Lemma 4.21. Let F be an XNF formula and $L \subseteq \mathbb{L}_n$ be an assignment such that there is an XLIN-refutation π of $F \cup \widehat{L}$. Then there is an XLIN-proof $\pi' \colon F \vdash_{\mathsf{XLIN}} 1 + L$ whose size is polynomial in $\mathrm{size}(1 + L)$ and $\mathrm{size}(\pi)$.

Proof. Let $\pi \colon F \cup \widehat{L} \vdash_{\mathsf{XLIN}} \emptyset$ and write $\pi = (H_1, \ldots, H_k)$. Line-by-line, construct a new proof π' from π as follows. If $H_i \in F$ is an initial premise, add two lines that deduce $H_i \cup (1 + L)$ using weak from the initial premise $H_i \in F$ to π'. If $H_i = \{\ell\} \in \widehat{L}$ is an initial premise, we add the two lines which deduce $H_i \cup (1 + L)$ using weak from the axiom $\{\ell, \ell + 1\}$ to π'. Thus, π' uses only the clauses from F as initial premises.

Now, if H_i arises as the conclusion of a rule of inference $\mathsf{ri} \in \{\mathsf{weak}, \mathsf{uc}\}$, then there is H_j with $j < i$ and $H_j \vdash_{\mathsf{ri}} H_i$. In the case $\mathsf{ri} = \mathsf{uc}$ this means $1 \in H_j$ and $H_j = H_i \cup \{1\}$, and we have $H_j \cup (1+L) = H_i \cup \{1\} \cup (1+L) \vdash_{\mathsf{uc}} H_i \cup (1 + L)$. Thus we can add the line $H_i \cup (1 + L)$ to π'. Similarly, if $\mathsf{ri} = \mathsf{weak}$, we get $H_j \subseteq H_i$ and therefore $H_j \cup (1 + L) \vdash_{\mathsf{weak}} H_i \cup (1 + L)$. In other words, we can add $H_i \cup (1 + L)$ to π'. Finally, if H_i arises as the conclusion of the rule of inference add, then there are H_{j_1}, H_{j_2} with $j_1 < i$, $j_2 < i$, and $H_{j_1}, H_{j_2} \vdash_{\mathsf{add}} H_i$. Now write $H_{j_1} = H'_{j_1} \cup \{\ell_1\}$, $H_{j_2} = H'_{j_2} \cup \{\ell_2\}$ and $H_i = H'_{j_1} \cup H'_{j_2} \cup \{\ell_1 + \ell_2\}$. Then we have

$$H'_{j_1} \cup \{\ell_1\} \cup (1 + L),\; H'_{j_2} \cup \{\ell_2\} \cup (1 + L) \;\vdash_{\mathsf{add}}\; H'_{j_1} \cup H'_{j_2} \cup \{\ell_1 + \ell_2\} \cup (1 + L)$$

i.e., we can add $H_i \cup (1 + L)$ to the proof π'.

After the entire XLIN-refutation π of $F \cup \widehat{L}$ is processed in this way, for every line H_i, the clause $H_i \cup (1 + L)$ is a line of π'. In particular, this means that π' contains the line $\emptyset \cup (1 + L)$, i.e., $\pi' \colon F \vdash_{\mathsf{XLIN}} 1 + L$. By construction, π' consists of $H_i \cup (1 + L)$ for each line H_i of π, all the $H_i \in F$, and at most length(π) Boolean axioms. This shows

$$\mathrm{size}(\pi') \;\leq\; \big(\mathrm{size}(\pi) + \mathrm{length}(\pi) \cdot \mathrm{size}(1 + L)\big) \;+\; \mathrm{size}(\pi) \;+\; \mathrm{length}(\pi) \cdot (2 \cdot \mathrm{size}(1 + L)).$$

Since length$(\pi) \leq$ size(π), we get a polynomial bound on size(π') as claimed. □

The construction in the proof is not optimal for the size of the proof. Instead of weakening all premises H_i from F, it suffices to weaken only the premises $\{\ell\} \in \widehat{L}$. Eventually, the final conclusion must then be weakened to $1 + L$. This also bounds the number of additional lines by the number of such premises in π.

Proposition 4.22. Let F be an XNF formula and $L \subseteq \mathbb{L}_n$ a valid σ-assignment.

(a) Let $\ell \in \mathrm{GCP}_\sigma(F, L)$. Then there is a polynomial-size XLIN-proof $F \cup \widehat{L} \vdash_{\mathsf{XLIN}} \{\ell\}$.

(b) If $F|_L$ is unsatisfiable, then there is an XLIN-proof $F \vdash_{\mathsf{XLIN}} 1 + L$ whose size is polynomial in size(F), size$(1 + L)$, and the number of decisions of XDPLL(F, L).

Proof. First we prove (a). By definition of GCP every element $\ell \in \mathrm{GCP}_\sigma(F, L)$ either belongs to L or there is a clause $C \in F$ and $L' \subseteq \mathrm{GCP}_\sigma(F, L) \setminus \{\ell\}$ with $C|_{L'} \equiv \{\ell\}$. Using induction on the size of L' and applying Remark 4.5.a, we see that there exists a polynomial-size proof $F \cup \widehat{L} \vdash_{\mathsf{XLIN}} \{\ell\}$.

To see claim (b), notice that due to $F|_L$ being unsatisfiable, all recursive calls of XDPLL(F, L) must eventually terminate in line 2 or 6. Whenever they terminate in line 2, we have $1 \in L' = \mathrm{GCP}(F, L)$ and by (a) and Lemma 4.21 there is a proof $F \vdash_{\mathsf{XLIN}} 1 + L$ whose size is polynomial in size(F) and size$(1 + L)$.

Otherwise, they terminate in line 6 and both recursive calls, XDPLL$(F, L' \cup \{\ell'\})$ in line 5 and XDPLL$(F, L' \cup \{\ell' + 1\})$ in line 6, return UNSAT. By induction, there are proofs of $F \vdash_{\mathsf{XLIN}} (1 + L') \cup \{\ell' + 1\}$ and of $F \vdash_{\mathsf{XLIN}} (1 + L') \cup \{\ell'\}$ each with size polynomial in size(F), size$(1 + L)$, and the number of decisions in the respective recursive XDPLL calls. By applying the rule of inference add to these two conclusions and using uc, we get an XLIN-proof $F \vdash_{\mathsf{XLIN}} L' + 1$. Moreover, its size is polynomial in size(F), size$(1 + L)$ and the number of decisions of XDPLL(F, L). □

In particular, if F is unsatisfiable, there is an XLIN-refutation of F whose size is polynomial in the running time of XDPLL(F). Additionally, XDPLL can be modified to output such a refutation with a polynomial-time overhead.

In order to boost the performance of XDPLL, the natural next step is to mimic conflict-driven clause learning methods, i.e., generalize CDCL to XNF formulae. The goal is to enhance the propagation step avoiding previously unsatisfiable search trees by learning clauses from every conflict. When the clauses are added, they are also used to *backjump* in the search tree so that the search can continue in a more focused manner. To reach this goal, we formalize a run of XDPLL with the following notions.

Definition 4.23. Let $d \in \mathbb{N}$.

(a) A tuple $\mathcal{L} = (L_0, \ldots, L_d)$ of σ-assignments with $L_0 \subsetneq L_1 \subsetneq \cdots \subsetneq L_d \subseteq \mathbb{L}_n$ is called a **σ-assignment stack** of size d. If the term order σ is clear from the context, we simply say $\mathcal{L}$ is an assignment stack. If $L_0, \ldots, L_d$ are variable assignments, then we call $\mathcal{L}$ a **variable assignment stack**.

(b) Let $\mathcal{L} = (L_0, \ldots, L_d)$ be an assignment stack. For $i \in \{0, \ldots, d\}$, the subspace $\langle L_i \rangle_{\mathbb{F}_2}$ denoted by $\mathcal{L}_i$ is called the **i-th assignment subspace**.

(c) For every lineral $\ell \in \mathcal{L}_d \cup (1 + \mathcal{L}_d)$, the integer

$$\delta_{\mathcal{L}}(\ell) \;=\; \min\big\{\, i \in \{0, \ldots, d\} \mid \ell \in \mathcal{L}_i \cup (1 + \mathcal{L}_i) \,\big\}$$

is called the **decision level** (or **implication level**) of ℓ.

Remark 4.24. Let F be an XNF formula. An assignment stack arises naturally from an execution of XDPLL(F) by denoting the assignment L' in the i-th level of recursion by L_i. Then, at all times, we have $L_0 \subsetneq L_1 \subsetneq \cdots \subsetneq L_d \subseteq \mathbb{L}_n$, and $\mathcal{L} = (L_0, \ldots, L_d)$ forms an assignment stack.

Next we capture the order in which assignments are constructed through GCP.

Definition 4.25. Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be a formula in XNF and $L \subseteq \mathbb{L}_n$ an assignment. A sequence of tuples $(\ell_1, C_1), \ldots, (\ell_k, C_k)$ in $(\mathbb{L}_n \setminus \{0\}) \times F$, where, for every $i \in \{1, \ldots, k\}$, the clause C_i is a reason clause for ℓ_i under $L \cup \{\ell_1, \ldots, \ell_{i-1}\}$, i.e., where

$$C_i|_{L \cup \{\ell_1, \ldots, \ell_{i-1}\}} \equiv \{\ell_i\},$$

is called a **lineral trail for** F **under** L. The lineral trail is also denoted by $[\ell_1^{C_1}, \ldots, \ell_k^{C_k}]$ or by $[\ell_1, \ldots, \ell_k]$, if the reason clauses are irrelevant in the respective context.

Remark 4.26. Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be a formula in XNF and $L \subseteq \mathbb{L}_n$ a σ-assignment. If $[\ell_1, \ldots, \ell_k]$ is a lineral trail for F under L, then $\{\ell_1, \ldots, \ell_k\} \subseteq \langle \mathrm{GCP}(F, L) \rangle_{\mathbb{F}_2}$. In particular, this shows $F|_L \models \{\ell_i\}$ for all $i \in \{1, \ldots, k\}$. Conversely, the computation of $L' = \mathrm{GCP}(F, L)$ gives rise to a lineral trail $[\ell_1, \ldots, \ell_k]$ for F under L, where $\{\ell_1, \ldots, \ell_k\} = L' \setminus L$.

Example 4.27. The execution of GCP in Example 4.17 yields the lineral trail

$$[\, x_3 + x_8^{C_1},\ x_8 + 1^{C_2},\ x_2 + x_4^{C_3},\ x_7^{C_4},\ x_4 + x_5 + x_6 + 1^{C_5},\ 1^{C_6} \,]$$

for F under $L = \{x_1 + x_2,\ x_5\}$.

A state of XDPLL is now given as an *assignment stack* which arises from *lineral trails* and *decisions*.

Definition 4.28. Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be an XNF formula and $\mathcal{L} = (L_0, L_1, \ldots, L_d)$ be a σ-assignment stack. The tuple $\mathcal{L}$ is called an (XNF) **solver state for** F with **decision sequence** $(\ell_0^{(1)}, \ldots, \ell_0^{(d)})$, if

(1) we have $L_0 = \{\ell_1^{(0)}, \ldots, \ell_{k_0}^{(0)}\}$ with a lineral trail $[\ell_1^{(0)}, \ldots, \ell_{k_0}^{(0)}]$ for F under $\emptyset$, and

(2) for $i > 0$, we have $L_i = L_{i-1} \cup \{\ell_0^{(i)}\} \cup \{\ell_1^{(i)}, \ldots, \ell_{k_i}^{(i)}\}$, where $\ell_0^{(i)}$ is a decision for A_{i-1} and $[\ell_1^{(i)}, \ldots, \ell_{k_i}^{(i)}]$ is a lineral trail for F under $L_{i-1} \cup \{\ell_0^{(i)}\}$.

For $i \geq 0$, the lineral trail $[\ell_1^{(i)}, \ldots, \ell_{k_i}^{(i)}]$ is called the i**-th lineral trail** of $\mathcal{L}$. If we additionally have $1 \in L_d$, then $\mathcal{L}$ is said to be a **conflict** (XNF) **solver state**.

By Remarks 4.24 and 4.26, every vertex in a search tree arising from a run of XDPLL can be described by an XNF solver state for F.

Example 4.29. Consider the XNF formula $F = \{C_1, C_2, C_3\} \subseteq \mathcal{P}(\mathbb{L}_5)$ with clauses

$$C_1 = \{x_1 + x_4,\ x_3 + x_4\},\ C_2 = \{x_2 + x_5 + 1,\ x_3 + x_5 + 1\},\ C_3 = \{x_2 + x_4 + 1,\ x_4 + x_5\}.$$

An execution of XDPLL(F) with a *lexicographic* decision heuristic yields the XNF conflict solver state $\mathcal{L} = (L_0, L_1, L_2, L_3)$, where

$$\begin{aligned} L_0 = \emptyset &\subseteq L_1 = L_0 \cup \{x_1\} \\ &\subseteq L_2 = L_1 \cup \{x_2\} \\ &\subseteq L_3 = L_2 \cup \{x_3,\ x_4,\ x_5 + 1,\ 1\}, \end{aligned}$$

with decision sequence $(x_1,\ x_2,\ x_3)$ and literal trails

$$\begin{aligned} &[\ ] && \text{for } F \text{ under } L_0 \cup \{x_1\}, \\ &[\ ] && \text{for } F \text{ under } L_1 \cup \{x_2\}, \\ &[x_4^{C_1},\ x_5 + 1^{C_2},\ 1^{C_3}] && \text{for } F \text{ under } L_2 \cup \{x_3\}. \end{aligned}$$

4.3 Learning Asserting XNF Clauses Using Lineral Trails

The key improvement of CDCL over DPLL is the *learning* of a new clause whenever the solver is in a conflict state. In the CNF setting, usually only *asserting* clauses are considered. These can be generalized as follows.

Definition 4.30. Let $\mathcal{L} = (L_0, \dots, L_d)$ be an assignment stack and let $C \subseteq \mathbb{L}_n$ be a conflict clause under L_d. The clause C is called **$\mathcal{L}$-asserting**, if $C|_{L_{d-1}} \not\equiv \bot$ is a unit clause. The integer

$$\beta_{\mathcal{L}}(C) \;=\; \min\bigl\{i \in \{0, \dots, d-1\} \mid C|_{L_i} \text{ is a unit clause}\bigr\}$$

is called the **assertion level** of C, and the lineral $\ell \in \mathbb{L}_n$ with $C|_{L_{\beta_{\mathcal{L}}(C)}} \equiv \{\ell\}$ is called the **asserted lineral** of C.

In CDCL, the computation of an asserting clause is realized by computing resolvents along the reason clauses of the *literal* trail of the highest decision level. The theoretic foundation for this approach is given by the fact that the resolvent of a *conflict* and a certain *reason clause* is another *conflict clause* but under a *smaller* assignment. This allows to iteratively remove literals from the highest decision level until none are left and an asserting clause is found. We must be more careful to mimic this approach with XNF formulae, as the following example shows.

Example 4.31. Consider the setting of Example 4.29. In the highest decision level, we have the lineral trail $[x_4^{C_1},\, x_5 + 1^{C_2},\, 1^{C_3}]$ for F under $\{x_1,\, x_2,\, x_3\}$. Following the core idea of CNF-based learning we notice that under $L' = \{x_1, x_2, x_3, x_4\}$ we have $C_2|_{L'} \equiv \{x_5 + 1\}$ and $C_3|_{L'} \equiv \{x_5\}$. From these two clauses, we want to deduce a new conflict clause C under L'.

(a) In the setting of CNF clauses, we simply compute the resolvent of C_2 and C_3. However, the clauses C_2 and C_3 share no lineral to resolve on, i.e., the resolvent is given by $R = C_2 \cup C_3$. Unfortunately,

$$1 \;\in\; \langle x_2 + x_5,\, x_3 + x_5\rangle_{\mathbb{F}_2} + \langle x_2 + x_4, x_4 + x_5 + 1\rangle_{\mathbb{F}_2} \;=\; V_R$$

shows $R \equiv \top$ is a tautology and is, in particular, not a conflict clause under L'.

(b) To fix this we can *rewrite* the clauses as $C_2' = \{x_2+x_3+1,\, x_2+x_5+1\} \equiv C_2$ and $C_3' = \{x_2+x_4+1,\, x_2+x_5\} \equiv C_3$. Resolving C_2' and C_3' on the lineral $x_2 + x_5$ yields now the clause $C = \{x_2 + x_3 + 1,\, x_2 + x_4 + 1\}$. As desired, we have $C|_{L'} \equiv \bot$ and C is a conflict clause under L'.

This example shows that resolvents are sensitive to the representation of the clauses, and a more sophisticated method that finds the correct representation is preferable. Ideally, it even works with the equivalence classes directly so that the chosen representation becomes irrelevant.

In a slightly different setting, we can combine two reason clauses to get a reason clause for the sum of their implied linerals as follows.

Proposition 4.32. Let $L \subseteq \mathbb{L}_n$ be a valid assignment and $u_1, u_2 \in \mathbb{L}_n \setminus \{0\}$ be distinct linerals. For $i \in \{1, 2\}$, let $C_i \subseteq \mathbb{L}_n$ be a reason clause for u_i under L with reason clause decomposition $V_{C_i} = \bigl(V_{C_i} \cap \langle L\rangle_{\mathbb{F}_2}\bigr) \oplus \langle \ell_i + 1\rangle_{\mathbb{F}_2}$. Then there is a reason clause $R \subseteq \mathbb{L}_n$ for $u_1 + u_2$ under L with a polynomial-size XLIN-proof $C_1, C_2 \vdash_{\mathsf{XLIN}} R$ and

$$V_R \;=\; \bigl(V_{C_1} \cap \langle L\rangle_{\mathbb{F}_2}\bigr) \;+\; \bigl(V_{C_2} \cap \langle L\rangle_{\mathbb{F}_2}\bigr) \;+\; \langle \ell_1 + \ell_2 + 1\rangle_{\mathbb{F}_2}.$$

Proof. By Remark 4.13 there is a clause C_i' such that $1 + C_i'$ consists of ℓ_i and a basis B_i of $V_{C_i} \cap \langle L\rangle_{\mathbb{F}_2}$, such that $C_i \equiv C_i'$, and such that there is a polynomial-size XLIN-proof $C_i \vdash_{\mathsf{XLIN}} C_i'$. Now consider $R = (1 + B_1) \cup (1 + B_2) \cup \{\ell_1 + \ell_2\}$. Then $C_1', C_2' \vdash_{\mathsf{add}} R$ implies the existence of a polynomial-size proof $C_1, C_2 \vdash_{\mathsf{XLIN}} R$. Moreover,

$$\begin{aligned} V_R \;&=\; \langle B_1\rangle_{\mathbb{F}_2} \;+\; \langle B_2\rangle_{\mathbb{F}_2} \;+\; \langle \ell_1 + \ell_2 + 1\rangle_{\mathbb{F}_2} \\ &=\; \bigl(V_{C_1} \cap \langle L\rangle_{\mathbb{F}_2}\bigr) \;+\; \bigl(V_{C_2} \cap \langle L\rangle_{\mathbb{F}_2}\bigr) \;+\; \langle \ell_1 + \ell_2 + 1\rangle_{\mathbb{F}_2}. \end{aligned}$$

By Proposition 4.8.a, R is a reason clause for $(\ell_1 + \ell_2)|_L \;=\; \ell_1|_L + \ell_2|_L \;=\; u_1 + u_2$ under L. □

Remark 4.33. In the setting of Proposition 4.32, it may seem that the choice of the reason linerals ℓ_1 and ℓ_2 can impact the associated subspace V_R of R. However, due to Remark 4.11, these reason linerals are unique *modulo* $V_{C_i} \cap \langle L \rangle_{\mathbb{F}_2}$, and by $V_{C_i} \cap \langle L \rangle_{\mathbb{F}_2} \subseteq V_R \cap \langle L \rangle_{\mathbb{F}_2}$ also $\ell_1 + \ell_2$ is unique *modulo* $V_R \cap \langle L \rangle_{\mathbb{F}_2}$. Thus, V_R is determined completely by C_1, C_2, and L, i.e., there are no *bad* nor *good* choices for the reason clause decompositions of V_{C_1} and V_{C_2}.

In the CNF setting, learning asserting clauses is relatively straightforward: literals are processed in the order they appear in the literal trail, and whenever one of them appears negated in the (current) conflict clause, we resolve this clause with the respective reason clause. As a result, this literal is removed entirely from the clause.

In the language of XNF a clause $C \subseteq \mathbb{L}_n$ is a conflict clause under an assignment $L \subseteq \mathbb{L}_n$ if and only if $V_C \subseteq \langle L \rangle_{\mathbb{F}_2}$ (see Proposition 4.8.b). Because of $1 + C \subseteq V_C$, the negation of every literal of C is contained in the *linear span* of L, but not necessarily in L itself. Consequently, multiple different linerals may need to be combined to form the negation of a lineral $\ell \in C$. In order to *eliminate* a lineral ℓ in C, we must remove all linerals from L that *occur* in ℓ one-by-one using Proposition 4.32. To formalize this idea, we introduce the following concept of *reducers*.

Definition 4.34. Let $L \subseteq \mathbb{L}_n$ be a valid σ-assignment and $\ell \in \mathbb{L}_n$. By Remark 4.3, the set L is an $\mathbb{F}_2$-basis of $\langle L \rangle_{\mathbb{F}_2}$ and $\ell|_L \in \ell + \langle L \rangle_{\mathbb{F}_2}$. Thus, there is a unique set $\mathrm{RS}_L(\ell) \subseteq L$ with $\ell|_L = \ell + \sum_{\ell' \in \mathrm{RS}_L(\ell)} \ell'$ which we call the **set of reducers** of ℓ with respect to L.

Remark 4.35. Let L be a valid σ-assignment and $\ell \in \mathbb{L}_n$. In general, the set $\mathrm{RS}_L(\ell)$ can be computed in polynomial time. If L is additionally σ-interreduced, i.e., for all distinct $\ell, \ell' \in L$, we have $\mathrm{LT}_\sigma(\ell) \notin \mathrm{Supp}(\ell')$, then we have $\mathrm{RS}_L(\ell) = \{\ell' \in L \mid \mathrm{LT}_\sigma(\ell') \in \mathrm{Supp}(\ell)\}$ which can be found in linear time.

Example 4.36.

(a) Consider the Lex-assignment $L = \{x_1+x_2+x_4,\ x_2+x_3+x_4,\ x_4+x_5\} \subseteq \mathbb{L}_6$. Then we have $\mathrm{RS}_L(x_2+x_5+x_6) = \{x_2+x_3+x_4,\ x_4+x_5\}$ and $\mathrm{RS}_L(x_1+x_3+x_4) = L$.

(b) Consider the Lex-assignment $L' = \{x_1+x_3,\ x_2+x_3+x_5,\ x_4+x_5\}$, which is Lex-interreduced and satisfies $\langle L \rangle_{\mathbb{F}_2} = \langle L' \rangle_{\mathbb{F}_2}$ Then we have $\mathrm{RS}_{L'}(x_2+x_5+x_6) = \{x_2+x_3+x_5\}$ and $\mathrm{RS}_{L'}(x_1+x_3+x_4) = \{x_1+x_3,\ x_4+x_5\}$.

Finally, we combine the above to yield our central tool for conflict learning. It allows us to remove one reducer of the reason lineral of a conflict clause using a corresponding reason clause.

Proposition 4.37. Let $L \subseteq \mathbb{L}_n$ be an assignment and $\ell \in \mathbb{L}_n$ be a lineral such that $L \cup \{\ell\}$ is a valid assignment. Let $C \subseteq \mathbb{L}_n$ be a conflict clause under $L \cup \{\ell\}$, but not under L, and let $R \subseteq \mathbb{L}_n$ be a reason clause for ℓ under L.

(a) Then C is a reason clause for $\ell + 1$ under L, and every reason lineral ℓ_C of C under L satisfies $\ell \in \mathrm{RS}_{L \cup \{\ell\}}(\ell_C)$.

(b) Let ℓ_C be a reason lineral of C under L and ℓ_R be a reason lineral of R under L. Then there is a conflict clause $C' \subseteq \mathbb{L}_n$ under L with a polynomial-size XLIN-proof $R, C \vdash_{\mathsf{XLIN}} C'$ and

$$V_{C'} = \big(V_C \cap \langle L \rangle_{\mathbb{F}_2}\big) + \big(V_R \cap \langle L \rangle_{\mathbb{F}_2}\big) + \langle \ell_C + \ell_R + 1 \rangle_{\mathbb{F}_2}.$$

Proof. First we show (a). By Proposition 4.8, we have $V_C \subseteq \langle L \cup \{\ell\} \rangle_{\mathbb{F}_2}$ and $V_C \not\subseteq \langle L \rangle_{\mathbb{F}_2}$. Then the canonical epimorphism $\varphi \colon V_C \to \langle L \cup \{\ell\} \rangle_{\mathbb{F}_2} / \langle L \rangle_{\mathbb{F}_2}$ is well-defined, and we have $\ker(\varphi) = V_C \cap \langle L \rangle_{\mathbb{F}_2}$. By the rank-nullity theorem, we get

$$\begin{aligned} \dim_{\mathbb{F}_2} V_C \,/\, \big(V_C \cap \langle L \rangle_{\mathbb{F}_2}\big) &= \dim_{\mathbb{F}_2} V_C - \dim_{\mathbb{F}_2} \ker(\varphi) = \dim_{\mathbb{F}_2} \mathrm{im}(\varphi) \\ &\leq \dim_{\mathbb{F}_2} \big(\langle L \cup \{\ell\} \rangle_{\mathbb{F}_2} / \langle L \rangle_{\mathbb{F}_2}\big) \leq 1. \end{aligned}$$

Thus $C|_L$ is a non-trivial unit clause, see Remark 4.9, i.e., C is a reason clause under L.

Let ℓ_C be a reason lineral of C under L with $V_C = \big(V_C \cap \langle L\rangle_{\mathbb{F}_2}\big) \oplus \langle \ell_C + 1\rangle_{\mathbb{F}_2}$. Then $V_C \not\subseteq \langle L\rangle_{\mathbb{F}_2}$ shows $\ell_C + 1 \notin \langle L\rangle_{\mathbb{F}_2}$, and $V_C \subseteq \langle L \cup \{\ell\}\rangle_{\mathbb{F}_2}$ implies $\ell_C + 1 \in \langle L \cup \{\ell\}\rangle_{\mathbb{F}_2}$. Hence $\ell \in \mathrm{RS}_{L\cup\{\ell\}}(\ell_C)$ is one of the reducers of ℓ_C. Using $\ell_C|_{L\cup\{\ell\}} = 1$, this yields $\ell_C|_L = \ell + 1$, i.e., C is a reason clause for $\ell + 1$ under L.

Part (b) is an immediate consequence of (a) and Proposition 4.32. □

This proposition allows us to generalize CNF-based conflict learning in a straightforward way, see Algorithm 3. It proceeds along the lineral trail and iteratively removes *reducers* from a lineral of the conflict clause. It stops as soon as there is only a single lineral from the current decision level, i.e., when we have an asserting conflict clause.

Algorithm 3: LinTrailRes – Clause Learning by Resolving along Lineral Trail

Input : A conflict XNF solver state $\mathcal{L} = (L_0, \dots, L_d)$ for an XNF formula F.
Output: An $\mathcal{L}$-asserting XNF clause $C \subseteq \mathbb{L}_n$ with $F \vdash_{\mathsf{XLIN}} C$.

1 Let ℓ_0 be the d-th decision and $[\ell_1^{C_1}, \dots, \ell_k^{C_k}, 1^C]$ be the d-th lineral trail of $\mathcal{L}$.
2 Let $V = \langle 1 + C\rangle_{\mathbb{F}_2}$ and $B = 1 + C$.
3 **for** $i = k$ **to** 1 **do**
4 Let $L' = L_{d-1} \cup \{\ell_0, \dots, \ell_{i-1}\}$.
5 **if** $V \not\subseteq \langle L'\rangle_{\mathbb{F}_2}$ **then**
6 Write $V = \big(V \cap \langle L'\rangle_{\mathbb{F}_2}\big) \oplus \langle \ell' + 1\rangle_{\mathbb{F}_2}$ for some $\ell' \in \mathbb{L}_n$.
7 Write $V_{C_i} = \big(V_{C_i} \cap \langle L'\rangle_{\mathbb{F}_2}\big) \oplus \langle \ell'_i + 1\rangle_{\mathbb{F}_2}$ for some $\ell'_i \in \mathbb{L}_n$.
8 Replace V by $(V \cap \langle L'\rangle_{\mathbb{F}_2}) + \big(V_{C_i} \cap \langle L'\rangle_{\mathbb{F}_2}\big) + \langle \ell' + \ell'_i + 1\rangle_{\mathbb{F}_2}$.
9 Let B be a basis of V.
10 **if** $1 + B$ *is* $\mathcal{L}$*-asserting* **then return** $1 + B$.
11 **return** $1 + B$.

Proposition 4.38. Let $\mathcal{L} = (L_0, \dots, L_d)$ be a conflict XNF solver state for an XNF formula F. Then LinTrailRes is an algorithm which returns an $\mathcal{L}$-asserting clause $C \subseteq \mathbb{L}_n$. Moreover, there exists a polynomial-size proof $F \vdash_{\mathsf{XLIN}} C$.

Proof. Denote the assignment L' of the i-th iteration of the loop (lines 3 to 10) by L'_{i-1}. Then we have $L'_i = L_{d-1} \cup \{\ell_0, \dots, \ell_i\}$. By induction, we show that after the i-th iteration of the loop we have: (1) $1 + B$ is a conflict clause under L'_{i-1}, (2) $1 + B$ contains at least one lineral from decision level d, and (3) there is a polynomial-size XLIN-proof $C_i, \dots, C_k, C \vdash_{\mathsf{XLIN}} 1 + B$.

Let $i \le k$ and assume that at the beginning of the loop $1 + B$ is a conflict clause under L'_i with a polynomial-size XLIN-proof from $C_{i+1}, \dots, C_k, C$. If $V \subseteq \langle L'_{i-1}\rangle_{\mathbb{F}_2}$, then $1 + B$ is a conflict clause under L'_{i-1} and B is not changed in this iteration. In case $V \not\subseteq \langle L'_{i-1}\rangle_{\mathbb{F}_2}$, then $1 + B$ not a conflict clause under L'_{i-1}, but under L'_i. By Proposition 4.37 the construction of V ensures that $1 + B$ becomes a conflict clause under L'_{i-1}. Additionally, the proposition guarantees the existence of a polynomial-size XLIN-proof of $1 + B$ from $C_i, \dots, C_k, C$. Furthermore, since C_i is not a reason clause on any prior decision level, $V_{C_i} \cap \langle L'_{i-1}\rangle_{\mathbb{F}_2} \subseteq V$ and thus $1 + B$ must contain at least one lineral from decision level d.

If the algorithm terminates in line 10, the correctness is clear. If it ends in line 11, $1 + B$ must be a conflict clause under $L'_0 = L_{d-1} \cup \{\ell_0\}$, i.e., $V \subseteq \langle L_{d-1} \cup \{\ell_0\}\rangle_{\mathbb{F}_2}$. Since ℓ_0 is the only lineral of decision level d in L'_0, we must have $\ell_0 \in V$, and $1 + B$ is an $\mathcal{L}$-asserting clause. □

Remark 4.39. To get an efficient implementation of LinTrailRes, the subspace $V \subseteq \mathbb{L}_n$ should be tracked by a basis B of V.

(a) The condition $V \not\subseteq \langle L' \rangle_{\mathbb{F}_2}$ in line 5 can be checked efficiently by asserting that $\mathrm{LT}_\sigma(\ell_i)$ occurs in a lineral of the basis B of V.

(b) The reason clause decomposition as in line 6 can be found, for instance, as follows: Write $B = B_0 \cup B_1$ with $B_1 = \{\ell \in B \mid \ell_i \in \mathrm{RS}_{L' \cup \{\ell_i\}}(\ell)\}$, then we have $B_0 \subseteq \langle L' \rangle_{\mathbb{F}_2}$. Now choose $\ell_1 \in B_1$, let $B_1' = \ell_1 + (B \setminus \{\ell_1\})$ and $B' = B_0 \cup B_1' \cup \{\ell_1\}$. Then $V = \langle B' \rangle_{\mathbb{F}_2}$ and $V \cap \langle L' \rangle_{\mathbb{F}_2} = \langle B_0 \cup B_1' \rangle_{\mathbb{F}_2}$. Hence the decomposition in line 6 is given by $V = \langle B_0 \cup B_1' \rangle_{\mathbb{F}_2} \oplus \langle \ell_1 + 1 \rangle_{\mathbb{F}_2}$.

Similarly, also the reason clause decomposition of V_{C_i} in line 7 can be found effectively by computing the linerals $\ell \in C_i$ with $\ell_i \in \mathrm{RS}_{L' \cup \{\ell_i\}}(\ell)$.

In general, such an implementation requires a polynomial runtime. This is in stark contrast to CDCL, where conflict learning can be implemented with a linear time runtime.

Observe that LinTrailRes is a direct generalization of its CNF analogue, as for CNF formulae lines 6-8 exactly compute the resolvent of the CNF clauses C_i and $C = 1 + B$.

Example 4.40. Let us compute $\mathsf{LinTrailRes}(\mathcal{L})$ for the conflict XNF solver state $\mathcal{L} = (L_0, L_1, L_2, L_3)$ from Example 4.29. For this, we proceed along the third lineral trail $[x_4^{C_1}, x_5 + 1^{C_2}, 1^{C_3}]$, and start with $V = V_{C_3}$ and $B = \{x_2 + x_4 + 1, x_4 + x_5 + 1\}$. Then we perform the following two loop iterations.

(1) Let $L' = \{x_1, x_2, x_3, x_4\}$. As $V \not\subseteq \langle L' \rangle_{\mathbb{F}_2}$ we need to update V and B. The reason clause decompositions of V and V_{C_2} are given by

$$\begin{array}{lcll} V & = & \langle x_2 + x_4,\ x_4 + x_5 + 1 \rangle_{\mathbb{F}_2} & = \langle x_2 + x_4 \rangle_{\mathbb{F}_2} \oplus \langle (x_4 + x_5) + 1 \rangle_{\mathbb{F}_2} \\ V_{C_2} & = & \langle x_2 + x_5,\ x_3 + x_5 \rangle_{\mathbb{F}_2} & = \langle x_2 + x_3 \rangle_{\mathbb{F}_2} \oplus \langle (x_2 + x_5 + 1) + 1 \rangle_{\mathbb{F}_2} \end{array}$$

and we update V such that we have

$$\begin{aligned} V &= \langle x_2 + x_4 \rangle_{\mathbb{F}_2} + \langle x_2 + x_3 \rangle_{\mathbb{F}_2} + \langle (x_4 + x_5) + (x_2 + x_5 + 1) + 1 \rangle_{\mathbb{F}_2} \\ &= \langle x_2 + x_4,\ x_2 + x_3 \rangle_{\mathbb{F}_2} \end{aligned}$$

with basis $B = \{x_2 + x_4,\ x_2 + x_3\}$.

The clause $1 + B$ is equivalent to the resolvent of the clauses C_2' and C_3' from Example 4.31.b, i.e., LinTrailRes changes the representation as intended.

(2) Let $L' = \{x_1, x_2, x_3\}$. As $V \not\subseteq \langle L' \rangle_{\mathbb{F}_2}$ we need to update V and B another time. The reason clause decompositions of V and V_{C_1} are given by

$$\begin{array}{lcll} V & = & \langle x_2 + x_4,\ x_2 + x_3 \rangle_{\mathbb{F}_2} & = \langle x_2 + x_3 \rangle_{\mathbb{F}_2} \oplus \langle (x_2 + x_4 + 1) + 1 \rangle_{\mathbb{F}_2} \\ V_{C_1} & = & \langle x_1 + x_4 + 1,\ x_3 + x_4 + 1 \rangle_{\mathbb{F}_2} & = \langle x_1 + x_3 \rangle_{\mathbb{F}_2} \oplus \langle (x_3 + x_4) + 1 \rangle_{\mathbb{F}_2} \end{array}$$

and we update V such that we have

$$\begin{aligned} V &= \langle x_2 + x_3 \rangle_{\mathbb{F}_2} + \langle x_1 + x_3 \rangle_{\mathbb{F}_2} + \langle (x_2 + x_4 + 1) + (x_3 + x_4) + 1 \rangle_{\mathbb{F}_2} \\ &= \langle x_2 + x_3,\ x_1 + x_3 \rangle_{\mathbb{F}_2} \end{aligned}$$

with basis $B = \{x_2 + x_3,\ x_1 + x_3\}$.

Finally, the $\mathcal{L}$-asserting clause $C = 1 + B = \{x_2 + x_3 + 1, x_1 + x_3 + 1\}$ is returned. Its assertion level is $\beta_{\mathcal{L}}(C) = 2$ and the asserted lineral is $C|_{L_2} \equiv \{x_3 + 1\}$.

4.4 The CDXCL Algorithm

After we presented a learning scheme for conflict XNF solver states, we are ready to add it to XDPLL, see CDXCL in Algorithm 4. The description of the algorithm differs substantially from XDPLL since we choose an iterative rather than a recursive formulation. This simplifies the description of the backtracking step and exemplifies how assignments contribute to different decision levels. It is a straightforward adaption of CDCL to XNF formulae, where we replace BCP by GCP and use LinTrailRes to learn asserting clauses.

Algorithm 4: CDXCL – Conflict-Driven XNF SAT Solving

Input : An XNF formula $F \subseteq \mathcal{P}(\mathbb{L}_n)$.
Output: UNSAT or $a \in \mathcal{S}(F)$.

1 Let $d = 0$, let $\Gamma = \emptyset$, and compute $L_0 = \mathrm{GCP}(F, \emptyset)$.
2 **if** L_0 *is invalid* **then return** UNSAT.
3 **while** L_d *is not a complete assignment* **do**
4 Increase d by one.
5 Choose a decision $\ell_0^{(d)}$ for L_{d-1}. `decision`
6 Compute $L_d = \mathrm{GCP}\big(F \cup \Gamma,\, L_{d-1} \cup \{\ell_0^{(d)}\}\big)$. `propagation`
7 **while** L_d *is invalid* **do**
8 **if** $d = 0$ **then return** UNSAT.
9 Add $C = \mathsf{LinTrailRes}\big(\,(L_0, \ldots, L_d)\,\big)$ to Γ. `conflict learning`
10 Let $d = \beta_{(L_0,\ldots,L_d)}(C)$ be the assertion level of C. `backtracking`
11 Compute $L_d = \mathrm{GCP}\big(F \cup \Gamma,\, L_d\big)$.
12 **return** $a \in \mathcal{Z}(L_d)$.

Proposition 4.41. Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be an XNF formula. Then, CDXCL is an algorithm. It returns UNSAT $=$ CDXCL(F) if and only if F is unsatisfiable. Otherwise, it returns a satisfying assignment $a \in \mathcal{S}(F)$.

Proof. Observe that at all times during the execution, the tuple $(L_0, \ldots, L_d)$ forms an XNF solver state for $F \cup \Gamma$. Moreover, by Proposition 4.38 we have $F \equiv F \cup \Gamma$. First, let us show that the algorithm terminates after finitely many steps. Whenever GCP is called in lines 6 and 11, we record a *dimension tuple* $(d_1, \ldots, d_n) \in \mathbb{N}^n$, where

$$d_i = \begin{cases} \dim_{\mathbb{F}_2} \langle L_i \rangle_{\mathbb{F}_2} & \text{if } i < d \\ \dim_{\mathbb{F}_2} \langle L_d \rangle_{\mathbb{F}_2} & \text{otherwise} \end{cases}, \text{ for } i \in \{1, \ldots, n\}.$$

Let us show that the sequence of dimension tuples is strictly increasing with respect to the lexicographical ordering on $\mathbb{N}^n$. Since $\ell_0^{(d)}$ is a decision for $\langle L_{d-1} \rangle_{\mathbb{F}_2}$ and $\ell_0^{(d)} \in L_d$ in line 5, we have $\dim_{\mathbb{F}_2} \langle L_d \rangle_{\mathbb{F}_2} > \dim_{\mathbb{F}_2} \langle L_{d-1} \rangle_{\mathbb{F}_2}$. This means that the entry in the d-th position of the dimension tuple increases strictly. The situation in line 11 is similar: Since an $\mathcal{L}$-asserting clause C is added to Γ and we backtrack to the assertion level of C, the call to GCP must find at least one *new* lineral. Hence, the dimension of L_d increases by at least one. Altogether, the dimension tuples form a strictly increasing sequence. Now it suffices to note that the entries of the dimension tuples are bounded from above by $\dim_{\mathbb{F}_2} \mathbb{L}_n = n + 1$, and L_0 is invalid in case $\dim_{\mathbb{F}_2} \langle L_0 \rangle_{\mathbb{F}_2} = n + 1$, i.e., the procedure terminates in line 8 at the latest.

Next, let us argue that the algorithm is correct. By construction, $F \models \Gamma$ and $\langle L_0 \rangle_{\mathbb{F}_2} = \langle \mathrm{GCP}(F \cup \Gamma, \emptyset) \rangle_{\mathbb{F}_2}$. This shows $\mathcal{S}(F) = \mathcal{S}(F) \cap \mathcal{S}(\Gamma) \cap \mathcal{Z}(L_0)$. Now, whenever $d = 0$, the algorithm terminates correctly in lines 2 and 8. Finally, assume that the algorithm terminates in line 12. In this case, L_d assigns all variables. Since we have

$L_d = \mathrm{GCP}(F \cup \Gamma, L_{d-1} \cup \{\ell_0^{(d)}\})$ (see line 6), there is no clause in $F \cup \Gamma$ that is violated by L_d, i.e., F is satisfied by L_d. □

Remark 4.42. Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be a formula in XNF.

(a) If F is unsatisfiable, then CDXCL can construct an XLIN-refutation of F. In particular, the size of this proof is polynomial in size(F) and the number of decisions of CDXCL(F). This is an immediate consequence of Propositions 4.38 and 4.22.a and the fact that the algorithm terminates with $1 \in \mathrm{GCP}(F \cup \Gamma, L_0)$.

(b) If the input formula F is a CNF formula, then a run of CDXCL(F) reasons only within RES and is equivalent to a run of CDCL(F).

(c) Note that CDXCL stays correct if we replace LinTrailRes in line 9 by any other *learning scheme* that constructs an $(L_0, \dots, L_d)$-asserting clause C with $F \models C$.

In particular, in line 9 we have $1 \in L_d \subseteq \langle \mathrm{GCP}(F \cup \Gamma, D) \rangle_{\mathbb{F}_2}$ for the set of decisions $D = \{\ell_0^{(0)}, \dots, \ell_0^{(d)}\}$. Now Proposition 4.22.a and Lemma 4.21 show that $F \models 1 + D$, and we can use a *learning scheme* that uses this $(L_0, \dots, L_d)$-asserting XNF clause $C = 1 + D$ in line 9. Its assertion level is $d - 1$ and we have $(1 + D)|_{L_{d-1}} = \{\ell_0^{(d)} + 1\}$, i.e., learning mimics the recursive calls of XDPLL.

Consequently, XDPLL can be seen as a special variant of CDXCL. This also proves Proposition 4.20.

The success of state-of-the-art SAT solving rests on several pillars. While CDCL serves as the core framework, selecting and developing lazy data structures, decision/restart/deletion heuristics, and a broad range of efficient pre-/inprocessing methods are key ingredients for practical solving, see (Audemard and Simon 2009; Balyo et al. 2014; Beame, Kautz, et al. 2004; Biere and Fröhlich 2018, 2015; Chu et al. 2010; Eén and Sörensson 2004; Gent 2013; Gomes et al. 1998; Hölldobler et al. 2010; Järvisalo, Biere, et al. 2010; Järvisalo, Heule, et al. 2012; Krüger et al. 2022; Lewis et al. 2005; Li et al. 2020; Lynce and Marques-Silva 2003; Marques-Silva and Sakallah 1996; Moskewicz et al. 2001; Nadel and Ryvchin 2018; Oh 2015; Pipatsrisawat and Darwiche 2007; Sörensson and Biere 2009). The same applies to CNF-XOR-based SAT solvers using $\mathrm{BCP}^{\mathrm{XOR}}_{\mathrm{GJE}}$, see (C.-S. Han and Jiang 2012; Laitinen et al. 2012b, 2010; Soos 2010; Soos, Gocht, et al. 2020; Soos, Nohl, et al. 2009). Despite the stronger propagation and learning schemes, there is no indication that the situation is fundamentally different for CDXCL in the context of XNF SAT solving. Thus, to compete with modern SAT solvers, these topics should also be addressed here.

Translating most restart and deletion heuristics to this more general setting is straightforward. For instance, the notion of *literal block distance*, initially proposed in (Audemard and Simon 2009), can be formulated as follows. Recall that we denote the i-th assignment subspace of an assignment stack $\mathcal{L}$ by $\mathcal{L}_i$.

Definition 4.43. Let $\mathcal{L} = (L_0, \dots, L_d)$ be an assignment stack. For an XNF clause $C \subseteq \mathcal{L}_d \cup (1 + \mathcal{L}_d)$, the **lineral block distance (LBD) of** C **under** $\mathcal{L}$ is given by

$$\mathrm{LBD}_{\mathcal{L}}(C) \;=\; \#\Big(\bigcup_{\ell \in C} \big\{\delta_{\mathcal{L}}(\ell') \mid \ell' \in \mathrm{RS}_{L_d}(\ell)\big\}\Big).$$

For a lineral $\ell \in \mathcal{L}_d \cup (1 + \mathcal{L}_d)$, the **lineral block distance (LBD) of** ℓ **under** $\mathcal{L}$ is given by $\mathrm{LBD}_{\mathcal{L}}(\ell) = \mathrm{LBD}_{\mathcal{L}}(\{\ell\})$.

Such a translation is possible for many other concepts from CNF-based SAT solving. However, the elephant in the room is finding adequate lazy data structures for an efficient implementation of CDXCL. Ideally, they should perform in the league of those for CDCL.

Remark 4.44. To obtain an efficient implementation of CDXCL, it is necessary to have a fast implementation of GCP, which supports efficient backtracking within CDXCL, as well as a fast implementation of the conflict learning routine LinTrailRes.

(a) Our propagation algorithm, GCP, requires us to determine all clauses which are a unit under a given lineral assignment $L \subseteq \mathbb{L}_n$, i.e., check whether a clause $C \subseteq \mathbb{L}_n$ satisfies $C|_L \equiv \{\ell\}$ for some $\ell \in \mathbb{L}_n$. There is no straightforward way to check this without explicitly computing $\ell'|_L$ for at least two distinct linerals $\ell' \in C$. Now, the computation of ℓ' under the lineral assignment L, i.e., under an LT_σ-interreduced set of linear polynomials, has a worst-case running time in $O(n^2)$. Moreover, if n is large and L contains many *long* linerals we have $\#\operatorname{Supp}(\ell'|_L) \gg \#\operatorname{Supp}(\ell')$. This shows that computing images under lineral assignments is *costly* and should be avoided whenever possible.

As a time-memory trade-off, we suggest storing the result of $\ell'|_L$ for each decision level as a starting point for the next decision level. Additionally, this allows easy backtracking by restoring the corresponding previous state. Note, however, that this has a significant impact on memory usage.

(b) The situation for conflict learning is similar; the core steps of LinTrailRes consist of checking whether a clause is a conflict under an assignment and finding reason clause decompositions if that is not the case. This essentially requires the computation of intersections of subspaces of $\mathbb{L}_n$. While these computations can be optimized slightly by caching information during the propagation, namely when a lineral is reduced to 1, still many reductions are required. A linear time complexity, as possible in the CNF setting, is clearly out of reach.

Unlike CDCL, an efficient implementation of CDXCL is difficult to achieve.

Last, but not least, in- and preprocessing methods should also be translated. While we deem this an important component of successful XNF SAT solving, it is beyond the scope of this paper. Instead, we focus on efficiently implementable modifications of CDXCL using a weaker propagation mechanism.

5 Effective Conflict-Driven XNF SAT Solving

In this section we optimize the trade-off between *deductive power* and *speed* of CDXCL. The primary motivation is that traditional CNF-based SAT solvers successfully employ *lazy data structures* which require only little memory interaction during propagation, clause learning, and backtracking. Thus, we introduce a new propagation mechanism for XNF SAT solving that allows efficient implementations at the cost of losing some of the deductive power.

5.1 Gaußian Constraint Propagation Lite

In particular, we change GCP so that only *assigning linerals*, i.e., literals, are propagated to the XNF clauses. To be more precise, we propagate a subset of the following set of *implied literals*.

Definition 5.1. Let $L \subseteq \mathbb{L}_n$ be a σ-assignment. Then we call

$$\operatorname{ImpLits}(L) \;=\; \left(\langle L\rangle_{\mathbb{F}_2} \setminus \{0\}\right) \cap \mathbb{X}_n$$

the set of **implied literals** of L.

To find this set of literals, one can use the σ-interreduction of L which we denote by $\mathrm{IR}_\sigma(L)$. It corresponds to a row-reduced echelon form of the linear polynomials in L where pivots are chosen using σ.

Proposition 5.2. Let $L \subseteq \mathbb{L}_n$ be a σ-assignment.

(a) If L is invalid, then $1 \in \mathrm{IR}_\sigma(L)$.

(b) If L is valid, then $\operatorname{ImpLits}(L) \;=\; \mathrm{IR}_\sigma(L) \cap \mathbb{X}_n$.

Proof. Note that $\mathrm{IR}_\sigma(L)$ is the reduced σ-Gröbner basis of $\langle L\rangle$. If $1 \in L$, then 1 is contained in every reduced Gröbner basis of $\langle L\rangle$, so in particular in $\mathrm{IR}_\sigma(L)$. This shows (a).

Now assume that L is a valid assignment, i.e., $1 \notin \langle L \rangle_{\mathbb{F}_2}$. Since we have $\mathsf{IR}_\sigma(L) \subseteq \langle L \rangle_{\mathbb{F}_2} \setminus \{0\}$, it suffices to show that $\mathrm{ImpLits}(L) \setminus \mathbb{F}_2 \subseteq \mathsf{IR}_\sigma(L) \cap \mathbb{X}_n$. Now let $x_i + v \in \mathrm{ImpLits}(L) \subseteq \langle L \rangle = \langle \mathsf{IR}_\sigma(L) \rangle$ with $v \in \mathbb{F}_2$ and $i \in \{1, \dots, n\}$ be a literal. As $\deg(x_i + v) = 1$, there are $\ell_1, \dots, \ell_k \in \mathsf{IR}_\sigma(L)$ with $x_i + v = \ell_1 + \cdots + \ell_k$. Because $\mathsf{IR}_\sigma(L)$ is σ-interreduced, this implies $\mathrm{LT}_\sigma(\ell_1), \dots, \mathrm{LT}_\sigma(\ell_k) \in \mathrm{Supp}(x_i + v) \subseteq \{x_i, 1\}$ and these leading terms are pairwise distinct. With $1 \notin \mathsf{IR}_\sigma(L)$ we get $k = 1$ and $x_i + v = \ell_1 \in \mathsf{IR}_\sigma(L) \cap \mathbb{X}_n$. □

Since computing *all* implied literals regularly is rather costly, we only use subsets of $\mathrm{ImpLits}(L)$ for propagation, most notably the set $L \cap \mathbb{X}_n$. For that purpose, consider the following modification of GCP, which uses only literals contained in a set of linerals $L \subseteq \mathbb{L}_n$ for propagation. Note that we do explicitly not require L to be a lineral assignment, i.e., it need not be LT_σ-interreduced.

Algorithm 5: $\mathrm{GCP}^{\mathrm{lite}}$ – Gaußian Constraint Propagation Lite

Input : An XNF formula $F \subseteq \mathcal{P}(\mathbb{L}_n)$, a set $L \subseteq \mathbb{L}_n$.
Output: A set $L' \subseteq \mathbb{L}_n$ with $L \subseteq L'$ and $\mathcal{S}(F) \cap \mathcal{Z}(L) = \mathcal{S}(F) \cap \mathcal{Z}(L')$.

1 **while** *there is a clause* $C \in F$ *s.t.* $C|_{L \cap \mathbb{X}_n} \equiv \{\ell\}$ *is a non-trivial unit clause*
2 | ***or*** *there is a lineral* $\ell' \in L$ *s.t.* $\ell'|_{L \cap \mathbb{X}_n} = \ell \in \mathbb{X}_n \setminus \{0\}$ *is a literal*
3 **do**
4 | Adjoin ℓ to L.
5 **return** L.

Remark 5.3.

(a) Let F be an XNF formula, let $L \subseteq \mathbb{L}_n$ be a set of linerals, and consider the σ-assignment $L_\sigma = \mathsf{IR}_\sigma(L)$ with $\langle L_\sigma \rangle_{\mathbb{F}_2} = \langle L \rangle_{\mathbb{F}_2}$. Then $L \cap \mathbb{X}_n \subseteq L_\sigma$ implies $\mathrm{GCP}^{\mathrm{lite}}(F, L) \subseteq \langle \mathrm{GCP}_\sigma(F, L_\sigma) \rangle_{\mathbb{F}_2}$.

Moreover, due to Propositions 4.22 and 4.32 there is a polynomial-size XLIN-proof $F \cup \widehat{L} \vdash_{\mathsf{XLIN}} \{\ell\}$ for every lineral $\ell \in \mathrm{GCP}^{\mathrm{lite}}(F, L)$.

(b) Let F be a CNF formula and $A \subseteq \mathbb{X}_n$ be a variable assignment. As there are only literals in F, the condition in line 2 never applies, we have $A \cap \mathbb{X}_n = A$, and we get $\mathrm{BCP}(F, A) = \mathrm{GCP}^{\mathrm{lite}}(F, A)$. Given Remark 4.15, this shows that $\mathrm{GCP}^{\mathrm{lite}}$ is another generalization of BCP.

Remember that XOR clauses are unit XNF clauses, see Remark 2.2. Now, there are CNF-XOR instances $F \subseteq \mathcal{P}(\mathbb{L}_n)$ and variable assignments $A \subseteq \mathbb{X}_n$, where we have $\mathrm{GCP}^{\mathrm{lite}}(F, A) \subsetneq \mathrm{BCP}^{\mathrm{XOR}}_{\mathrm{GJE}}(F, A)$ since the latter also searches for literals implied by the XOR clauses of $F|_A$ due its (lazy) Gauß-Jordan elimination mechanism. This naturally results in the following extension.

Remark 5.4 (Gauß-Jordan Elimination)**.** Consider the setting of $\mathrm{GCP}^{\mathrm{lite}}$. In order to improve the propagation power, denote the set of linerals which appear as unit clauses in F by $F_{\mathtt{unit}}$. Then extend the disjunctive condition in line 2 with

or *there is a lineral* $\ell' \in \langle F_{\mathtt{unit}} \rangle_{\mathbb{F}_2}$ *s.t.* $\ell'|_{L \cap \mathbb{X}_n} = \ell$ *is a literal.*

This is equivalent to checking if there are literals ℓ in $\langle F_{\mathtt{unit}}|_{L \cap \mathbb{X}_n} \rangle_{\mathbb{F}_2} \cap \mathbb{X}_n$. Denote this modified variant of $\mathrm{GCP}^{\mathrm{lite}}$ by $\mathrm{GCP}^{\mathrm{lite}}_{\mathrm{GJE}}$. For every $L \subseteq \mathbb{L}_n$, algorithm $\mathrm{GCP}^{\mathrm{lite}}_{\mathrm{GJE}}$ deduces (among others) *all* variables uniquely determined by the linear equations in $F_{\mathtt{unit}}|_{L \cap \mathbb{X}_n}$. This is essentially equivalent to performing a *Gauß-Jordan Elimination*, i.e., computing a row-reduced echelon form of the corresponding linear polynomials.

Note that $\mathrm{GCP}^{\mathrm{lite}}_{\mathrm{GJE}}$ proceeds exactly as $\mathrm{BCP}^{\mathrm{XOR}}_{\mathrm{GJE}}$ for CNF-XOR input formulae, see the respective constructions in (Soos 2010; Soos, Nohl, et al. 2009). Additionally, for GCP every lineral in $F_{\mathtt{unit}}$ is added to L, and therefore

additional literals that are found with the above modification must also lie in $\langle \mathrm{GCP}(F, A)\rangle_{\mathbb{F}_2}$ as it contains $F_{\mathtt{unit}}$ and $L \cap \mathbb{X}_n$.

Next, let us summarize the relations between the different propagation methods for CNF, CNF-XOR, and XNF formulae.

Remark 5.5. By Remarks 4.15, 5.3 and 5.4, $\mathrm{GCP}^{\mathsf{lite}}$, $\mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}$ and GCP are generalizations of BCP for CNF formulae. The latter two also generalize $\mathrm{BCP}^{\mathsf{XOR}}_{\mathsf{GJE}}$ for CNF-XOR formulae. Altogether, GCP is the strongest propagation mechanism. This can also be depicted somewhat informally as

$$\begin{aligned}
\text{CNF}: &\quad \mathrm{BCP} = \mathrm{BCP}^{\mathsf{XOR}}_{\mathsf{GJE}} = \mathrm{GCP}^{\mathsf{lite}} = \mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}} = \mathrm{GCP},\\
\text{CNF-XOR}: &\quad \mathrm{GCP}^{\mathsf{lite}} \cap \mathbb{X}_n \subseteq \mathrm{BCP}^{\mathsf{XOR}}_{\mathsf{GJE}} = \mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}} \cap \mathbb{X}_n \subseteq \langle \mathrm{GCP}\rangle_{\mathbb{F}_2},\\
\text{XNF}: &\quad \mathrm{GCP}^{\mathsf{lite}} \subseteq \mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}} \subseteq \langle \mathrm{GCP}\rangle_{\mathbb{F}_2},
\end{aligned}$$

where all containments are strict for appropriate input formulae.

Remark 5.6. Let $A \subseteq \mathbb{X}_n$ be a variable assignment and let F be an XNF formula. Then there is an CNF-XOR encoding G of F with

$$\mathrm{BCP}^{\mathsf{XOR}}_{\mathsf{GJE}}(G, A) \cap \mathbb{X}_n = \mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}(G, A) \cap \mathbb{X}_n = \mathrm{ImpLits}\big(\mathrm{GCP}_\sigma(F, A)\big).$$

A technical proof of this statement can be found in Appendix B.

This implies that appropriate encodings allow $\mathrm{BCP}^{\mathsf{XOR}}_{\mathsf{GJE}}$ and $\mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}$ to attain the full propagation power of GCP for variable assignments. The catch of the encoding, however, is that it comes with an exponential increase in the number of clauses. In the context of CNF-XOR solvers, it remains to be seen by experiment whether this trade-off between propagation *power* and *speed* is worthwhile.

For XNF based solvers, changing the input formula from XNF to CNF-XOR also heavily impacts the lineral trails. This affects both the speed of learning and the quality of the learnt clauses. Thus, we did not investigate this approach further. Instead, we consider the following suggestions to harness some more power of GCP.

Remark 5.7. Let F be an XNF formula and L be a lineral assignment. In most situations, the number of literals $\#(L \cap \mathbb{X}_n)$ which are used for propagation in $\mathrm{GCP}^{\mathsf{lite}}$ and $\mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}$ is much smaller than the number of implied literals $\#\,\mathrm{ImpLits}(L)$. So, to improve the power of propagation, we try to increase the number of elements in $L \cap \mathbb{X}_n$ in two different ways.

(a) A straightforward way is to explicitly calculate $\mathrm{ImpLits}(L)$ using Proposition 5.2 every once in a while and adjoin it to L. This, however, should not be done too often due to the computational overhead.

(b) Alternatively, heuristically *reduce* newly derived non-literals $\ell \in \mathbb{L}_n \setminus \mathbb{X}_n$ in line 4 with selected elements from L. In particular, we propose to compute the normal remainder of ℓ with respect to the set of **equivalence linerals** from L, i.e., linerals $\ell' \in L \setminus \mathbb{X}_n$ with $\#\,\mathrm{Vars}(\ell') = 2$. In this way, the number of variables $\#\,\mathrm{Vars}(\ell)$ of ℓ decreases or stays fixed during reduction.

Both suggestions can improve the *deductive reasoning* of $\mathrm{GCP}^{\mathsf{lite}}$ and $\mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}$, without changing the core idea of propagating only literals to the XNF clauses.

It turns out that XNF encodings of cryptographic problems often feature many short linerals, see (Andraschko et al. 2024), so the lineral assignments may contain many equivalence linerals. Hence, the heuristic reductions mentioned above may indeed make a huge difference.

To formalize the execution of $\mathrm{GCP}^{\mathsf{lite}}$ (and $\mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}$) and prepare for our learning scheme, we use the following modified notions of *lineral assignments* and *lineral trails*.

Definition 5.8. Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be a formula in XNF and $L \subseteq \mathbb{L}_n$ be a set of linerals.

(a) If, for every non-literal $\ell \in L \setminus \mathbb{X}_n$, there is a set $A \subseteq L \cap \mathbb{X}_n$ and a reason clause $C \in F$ for ℓ under A, then L is called a **lite lineral assignment for** F.

(b) A sequence of tuples $(\ell_1, C_1), \ldots, (\ell_k, C_k)$ in $(\mathbb{L}_n \setminus \{0\}) \times F$, where, for every $i \in \{1, \ldots, k\}$, the clause $C_i \in F$ is a reason clause for ℓ_i under the assignment $\big(L \cup \{\ell_1, \ldots, \ell_{i-1}\}\big) \cap \mathbb{X}_n$, i.e., where

$$C_i|_{(L \cup \{\ell_1, \ldots, \ell_{i-1}\}) \cap \mathbb{X}_n} \equiv \{\ell_i\},$$

is called a **lite lineral trail for** F **under** L. The lite lineral trail is also denoted by $[\ell_1^{C_1}, \ldots, \ell_k^{C_k}]$, or by $[\ell_1, \ldots, \ell_k]$, if the reason clauses are not of relevance in the respective context.

(c) Let $\mathcal{T} = [\ell_1, \ldots, \ell_k]$ be a lite lineral trail for F under L. Let $i_1 < \cdots < i_{k'}$ with $\{\ell_{i_1}, \ldots, \ell_{i_{k'}}\} = \{\ell_1, \ldots, \ell_k\} \cap \mathbb{X}_n$. Then $[\ell_{i_1}, \ldots, \ell_{i_{k'}}]$ is a *lite lineral trail for* F *under* $L \cap \mathbb{X}_n$ and we call it the **associated literal trail** of $\mathcal{T}$.

The associated *literal* trail arises from a lite *lineral* trail as the subsequence of literals. As expected, if L is a lite lineral assignment for a formula F, then $L' = \mathsf{GCP}^{\mathsf{lite}}(F, L)$ is also a lite lineral assignment for F.

Remark 5.9. Let $L \subseteq \mathbb{L}_n$ be a lite lineral assignment for an XNF formula F.

(a) Then the computation of $L' = \mathsf{GCP}^{\mathsf{lite}}(F, L)$ gives rise to a lite lineral trail $[\ \ell_1^{C_1}, \ldots, \ell_k^{C_k}\ ]$ for F under L, where $L' \setminus L = \{\ell_1, \ldots, \ell_k\}$.

For $i \in \{0, \ldots, k\}$, let $A_i = \big(L \cup \{\ell_1, \ldots, \ell_i\}\big) \cap \mathbb{X}_n$. If ℓ_i originates from a clause $C \in F$ with $C|_{A_{i-1}} \equiv \{\ell_i\}$, see line 1, then we let $C_i = C$. Now consider the case that ℓ_i originates from another lineral $\ell' \in L \cup \{\ell_1, \ldots, \ell_{i-1}\}$ with $\ell'|_{A_i} = \ell_i$, see line 2. Then, by construction, there is a set $A \subseteq A_{i-1}$ and a reason clause $C \in F$ with $C|_A \equiv \{\ell'\}$. Thus, $C|_{A_{i-1}} \equiv \{\ell'|_{A_{i-1}}\} = \{\ell_i\}$, and $C_i = C$ is a reason clause of ℓ_i under A_{i-1} as desired.

(b) If we allow to extend the formula F with new clauses C, where $F \vdash_{\mathsf{XLIN}} C$, then the computation of $L' = \mathsf{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}(F, L)$ can also produce such a lite lineral trail.

For every ℓ_i which is added to L' because of the extended condition in line 2 (see Remark 5.4), there is a lineral $\ell' \in \langle F_{\mathtt{unit}} \rangle_{\mathbb{F}_2}$ with $\ell'|_{A_{i-1}} = \ell_i$, i.e., $C_i = \{\ell'\}$ is a reason clause for ℓ_i under A_{i-1}. Clearly, there is also a polynomial-size proof $F \vdash_{\mathsf{XLIN}} \{\ell'\}$.

(c) The reduction heuristic of Remark 5.7.b requires a similar adjustment in order to provide a lite lineral trail.

Here it suffices to discuss the situation where a lineral ℓ in line 4 is reduced with a lineral $\ell' \in L \cup \{\ell_1, \ldots, \ell_{i-1}\}$. By construction, there are reason clauses $C, C' \in F$ with $C|_{A_{i-1}} \equiv \{\ell\}$ and $C'|_{A_{i-1}} \equiv \{\ell'\}$. By Proposition 4.32, we can explicitly construct a reason clause $C_i \subseteq \mathbb{L}_n$ for $\ell_i = \ell + \ell'$ under A_{i-1}. Moreover, there is a polynomial-size proof $F \vdash_{\mathsf{XLIN}} C_i$.

(d) The changes suggested in Remark 5.7.a can be handled analogously.

Whenever a literal $\ell_i \in \mathrm{ImpLits}(L \cup \{\ell_1, \ldots, \ell_{i-1}\})$ is added to L, then there is a subset $S \subseteq L \cup \{\ell_1, \ldots, \ell_{i-1}\}$ with $\ell_i = \sum_{\ell' \in S} \ell'$. Now, by construction, for each $\ell' \in S \setminus \mathbb{X}_n$ there is a subset $A \subseteq A_{i-1}$ and a reason clause for ℓ' under A. Similar to the above, Proposition 4.32 yields a reason clause C_i for

$$\sum_{\ell' \in S \setminus \mathbb{X}_n} \ell'|_{A_{i-1}} \ = \ \ell|_{A_{i-1}} \ = \ \ell$$

under A_{i-1}, along with a polynomial-size proof $F \vdash_{\mathsf{XLIN}} C_i$.

Altogether, there is a set Γ of XNF clauses such that an execution of $\mathsf{GCP}^{\mathsf{lite}}$ and $\mathsf{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}$ with and without the modifications of Remark 5.7 yield a lite lineral trail for $F \cup \Gamma$ under L. Moreover, for each clause $C \in \Gamma$, there is a polynomial-size proof $F \vdash_{\mathsf{XLIN}} C$.

The lineral trails are used only in the context of clause learning. Thus, the computation of reason clauses in Γ can be delayed until they are actually used during conflict analysis. To that effect, for each implied lineral ℓ, we store a tuple of clauses and linerals which need to be combined in the spirit of Proposition 4.32 to obtain the actual reason clause of ℓ.

Let us illustrate by example the hierarchy of XNF propagation mechanisms from Remark 5.5 and present corresponding lite lineral trails.

Example 5.10. Consider the setting of Example 4.17, i.e., let $L = \{x_1 + x_2,\ x_5\}$, and consider the formula F with clauses

$$\begin{aligned} C_1 &= \{x_5 + 1,\ x_3 + x_8\}, & C_2 &= \{x_3 + 1,\ x_5 + 1\},\\ C_3 &= \{x_3,\ x_1 + x_4\}, & C_4 &= \{x_3 + x_5,\ x_2 + x_4 + x_7\},\\ C_5 &= \{x_1 + x_3 + x_5 + x_6,\ x_1 + x_3 + x_4\}, & C_6 &= \{x_1 + x_5 + x_6,\ x_2 + x_4 + x_7 + 1\}. \end{aligned}$$

To ensure that L is a lite lineral assignment for F, we extend the formula by the XNF clause $C_7 = \{x_1 + x_2,\ x_5 + 1\}$.

(a) The computation of $L' = \mathrm{GCP}^{\mathrm{lite}}(F, L)$ can proceed as in Figure 2.a and return

$$L' = \{x_1 + x_2,\ x_5,\ x_3 + x_8,\ x_3 + 1,\ x_8 + 1,\ x_1 + x_4,\ x_2 + x_4 + x_7\}.$$

In particular, we obtain the subset of literals $L' \cap \mathbb{X}_n = \{x_5,\ x_3 + 1,\ x_8 + 1\}$.

This is in stark contrast to propagation with GCP, see Example 4.17, where an invalid assignment is derived, i.e., we learn that $F|_L$ is, in fact, unsatisfiable.

By Remark 5.9 this execution of $\mathrm{GCP}^{\mathrm{lite}}$ produces the following lite lineral trail for F under L, where the associated literal trail is highlighted:

$$[\, x_3 + x_8^{C_1},\ x_3 + 1^{C_2},\ x_8 + 1^{C_1},\ x_1 + x_4^{C_3},\ x_2 + x_4 + x_7^{C_4} \,].$$

(b) With the modifications of Remark 5.7.b an execution of $\mathrm{GCP}^{\mathrm{lite}}$ may proceed as in Figure 2.b, and deduce the lite lineral assignment

$$L' = \{x_1 + x_2,\ x_5,\ x_3 + x_8,\ x_3 + 1,\ x_8 + 1,\ x_2 + x_4,\ x_7\}$$

including a larger subset of literals $L' \cap \mathbb{X}_n = \{x_5,\ x_3 + 1,\ x_8 + 1,\ x_7\}$.

By Remark 5.9 we obtain the lite lineral trail

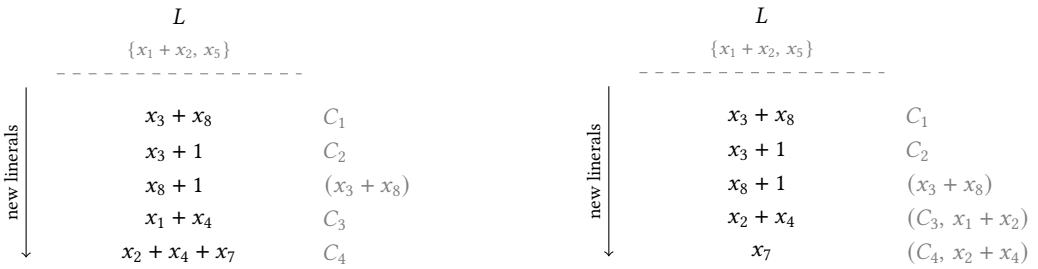
$$[\, x_3 + x_8^{C_1},\ x_3 + 1^{C_2},\ x_8 + 1^{C_1},\ x_2 + x_4^{C'_3},\ x_7^{C'_4} \,],$$

for $F \cup \Gamma$ under L, where we compute Γ using Proposition 4.32 as follows. The clause $C'_3 = \{x_3,\ x_5 + 1,\ x_2 + x_4\}$ arises from C_3 and the reason clause C_7 of $x_1 + x_2$, and $C'_4 = \{x_3,\ x_5 + 1,\ x_7\}$ arises from C_4 and the reason clause C'_3 of $x_2 + x_4$.

5.2 Conflict-Driven XNF SAT Solving Lite

Before we present our algorithm for effective XNF SAT solving, let us introduce the *states* which capture its execution. Note that, as we only propagate literals, i.e., variable assignments, to the XNF clauses in $\mathrm{GCP}^{\mathrm{lite}}$, every learnt clause should also propagate under some variable assignment to benefit our propagation mechanism. Consequently, all reason clauses need to propagate under variable assignments, and in particular, we only allow literal decisions.

Definition 5.11. Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be an XNF formula. Let $\mathcal{L} = (L_0, L_1, \ldots, L_d)$ with $L_0 \subseteq \cdots \subseteq L_d \subseteq \mathbb{L}_n$, and write $A_i = L_i \cap \mathbb{X}_n$, for $i \in \{0, \ldots, d\}$.



(a) Execution of $\mathrm{GCP}^{\mathrm{lite}}$.

(b) Execution of $\mathrm{GCP}^{\mathrm{lite}}$ using reductions with equivalence linerals (see Remark 5.7.b).

Fig. 2. Execution of $\mathrm{GCP}^{\mathrm{lite}}$ with the inputs from Example 5.10.

(a) Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be an XNF formula and $\mathcal{L} = (L_0, L_1, \ldots, L_d)$ be a σ-assignment stack. The tuple $\mathcal{L}$ is called a **lite** (XNF) **solver state for** F with **decision sequence** $(\ell_0^{(1)}, \ldots, \ell_0^{(d)})$, if

(1) $L_0 = \{\ell_1^{(0)}, \ldots, \ell_{k_0}^{(0)}\}$ with a lite lineral trail $[\ell_1^{(0)}, \ldots, \ell_{k_0}^{(0)}]$ for F under $\emptyset$, and

(2) for $i > 0$, we have $L_i = L_{i-1} \cup \{\ell_0^{(i)}\} \cup \{\ell_1^{(i)}, \ldots, \ell_{k_i}^{(i)}\}$, where $\ell_0^{(i)}$ is a decision for A_{i-1} and $[\ell_1^{(i)}, \ldots, \ell_{k_i}^{(i)}]$ is a lite lineral trail for F under $L_{i-1} \cup \{\ell_0^{(i)}\}$.

For $i \geq 0$, the lite lineral trail $[\ell_1^{(i)}, \ldots, \ell_{k_i}^{(i)}]$ is called the ***i*-th lite lineral trail** of $\mathcal{L}$. If we have $1 \in L_d$, then $\mathcal{L}$ is said to be a **lite conflict** (XNF) **solver state**.

(b) An assignment stack $\mathcal{A} = (A_0, \ldots, A_d)$ is called **stepwise** if, for all $i \in \{1, \ldots, d\}$, we have $\#(A_i \setminus A_{i-1}) = 1$ and $A_0 = \emptyset$.

(c) Let $\mathcal{L}$ be a lite solver state as in (a). Then the assignment stack $\mathcal{A} = (A_0, \ldots, A_d)$ is called the (associated) **variable assignment stack** of $\mathcal{L}$. For $i \geq 0$, the associated literal trail $[\ell_1^{(i)}, \ldots, \ell_{t_i}^{(i)}]$ of the i-th lite lineral trail is called the ***i*-th literal trail** of $\mathcal{L}$. The tuple

$$\mathcal{A}' = (\emptyset, A_{0,0}, \ldots, A_{0,t_0}, \ \ldots \ , A_{d,0}, \ldots, A_{d,t_d})$$

with $A_{i,t} = A_{i-1} \cup \{\ell_0^{(i)}, \ldots, \ell_{t_i}^{(i)}\}$ is called the (associated) **stepwise variable assignment stack** of $\mathcal{L}$.

Remark 5.12. Let $\mathcal{L} = (L_0, \ldots, L_d)$ be a lite XNF solver state for an XNF formula F.

(a) In contrast to the definition of *XNF solver states* (Definition 4.28), the tuple $\mathcal{L}$ is not a σ-assignment stack itself, i.e., the sets L_i need not be LT_σ-interreduced, but they are *lite lineral assignments* for F (Definition 5.8) by construction.

(b) The associated variable assignment stack $\mathcal{A}$ of $\mathcal{L}$ is clearly a variable assignment stack. Moreover, $\mathcal{A}$ is an *XNF solver state* for F, whose *lineral trails* contain only literals.

(c) Observe that stepwise variable assignment stacks arise naturally from lite lineral trails by considering their associated literal trails. In particular, the *associated* stepwise variable assignment stack simply concatenates these stepwise variable assignment stacks for the literal trails of all levels.

Example 5.13. The lite lineral trail $[\, x_3 + x_8,\ x_3 + 1,\ x_8 + 1,\ x_2 + x_4,\ x_7 \,]$ from Example 5.10.b yields the stepwise variable assignment stack corresponding to the chain

$$\emptyset \ \subset \ \{x_3 + 1\} \ \subset \ \{x_3 + 1,\ x_8 + 1\} \ \subset \ \{x_3 + 1,\ x_8 + 1,\ x_7\}.$$

Based on CDXCL, Algorithm 6 shows our framework for effective XNF SAT solving.

Algorithm 6: CDXCL$^{\text{lite}}$ – Conflict-Driven XNF SAT Solving Lite

Input : An XNF formula $F \subseteq \mathcal{P}(\mathbb{L}_n)$.
Output: UNSAT or $a \in \mathcal{S}(F)$.

1 Let $d = 0$, let $\Gamma = \emptyset$, compute $L_0 = \mathrm{GCP}^{\text{lite}}(F, \emptyset)$ and $A_0 = L_0 \cap \mathbb{X}_n$.
2 **if** L_0 *is invalid* **then return** UNSAT.
3 **while** A_d *is not a complete assignment* **do**
4 Increase d by one.
5 Choose a decision $\ell_0^{(d)} \in \mathbb{X}_n$ for A_{d-1}. `decision`
6 Compute $L_d = \mathrm{GCP}^{\text{lite}}\big(F \cup \Gamma, L_{d-1} \cup \{\ell_0^{(d)}\}\big)$ and $A_d = L_d \cap \mathbb{X}_n$. `propagation`
7 **while** A_d *is invalid* **do**
8 **if** $d = 0$ **then return** UNSAT.
9 Add $C = \mathsf{LinTrailRes}\big((A_0, \dots, A_d)\big)$ to Γ. `conflict learning`
10 Let $d = \beta_{(A_0,\dots,A_d)}(C)$ be the assertion level of C. `backtracking`
11 Compute $L_d = \mathrm{GCP}^{\text{lite}}(F \cup \Gamma, L_d)$ and $A_d = L_d \cap \mathbb{X}_n$.
12 **return** $a \in \mathcal{Z}(L_d)$.

Proposition 5.14. Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be an XNF formula. Then, CDXCL$^{\text{lite}}$ is an algorithm. It returns UNSAT = CDXCL$^{\text{lite}}(F)$ if and only if F is unsatisfiable. Otherwise, it returns a satisfying assignment $a \in \mathcal{S}(F)$.

Proof. Termination and correctness of the procedure follow analogous to Proposition 4.41. To see this, note that by Remark 5.12.b the tuple $(A_0, \dots, A_d)$ in line 9 is a conflict XNF solver state, and use the *dimension tuple* $(d_1, \dots, d_n) \in \mathbb{N}^n$ with

$$d_i \;=\; \begin{cases} \dim_{\mathbb{F}_2}\langle L_i\rangle_{\mathbb{F}_2} + \#A_i & \text{if } i < d \\ \dim_{\mathbb{F}_2}\langle L_d\rangle_{\mathbb{F}_2} + \#A_d & \text{otherwise} \end{cases}, \text{ for } i \in \{1, \dots, n\},$$

which is bounded from above by $2(n+1)$. □

Remark 5.15. Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be an XNF formula.

(a) An execution of CDXCL$^{\text{lite}}(F)$ can be characterized by a sequence of lite XNF solver states for $F \cup \Gamma$, since the tuple $(L_0, \dots, L_d)$ forms a lite XNF solver state after every call to GCP$^{\text{lite}}$ in lines 6 and 11.

(b) If F is unsatisfiable, then CDXCL$^{\text{lite}}$ can construct an XLIN-refutation of F. In particular, the size of this proof is polynomial in size(F) and the number of decisions of CDXCL$^{\text{lite}}(F)$. This is an immediate consequence of Propositions 4.38 and 4.22.a

(c) If the input formula F is a CNF formula, then a run of CDXCL$^{\text{lite}}(F)$ reasons only within RES and is equivalent to a run of CDCL(F).

(d) The algorithm stays correct if we extend GCP$^{\text{lite}}$ with the modifications of Remark 5.7, or replace it with GCP$^{\text{lite}}_{\text{GJE}}$. Denote the latter variant by CDXCL$^{\text{lite}}_{\text{GJE}}$.

In particular, CDXCL$^{\text{lite}}$ is another generalization of CDCL.

Remark 5.16. The condition in line 3 can be replaced by

not all clauses of F are satisfied by the partial assignment A_d.

This can also be checked efficiently by tracking the number of satisfied clauses in each decision level. As soon as this number is equal to $\#F$, all clauses are satisfied by A_d and, if $\mathcal{Z}(L_d) \neq \emptyset$, a solution can be returned in line 12. Otherwise, we get $1 \in \mathrm{ImpLits}(L_d)$, and a reason clause of this fact can be constructed analogous to Remark 5.9.c. After adding this clause to Γ, we simply continue from line 6.

This effectively avoids taking repeated decisions whenever the remaining, i.e., not yet satisfied part, of F is just a linear system of equations.

A consequence of the modified propagation mechanism is that many heuristics for CNF-based SAT solving can be transferred and used directly with CDXCL$^{\mathsf{lite}}$. This includes decision, deletion, and restart heuristics as follows.

Remark 5.17 (Heuristics)**.**

(a) Since we allow only literal decisions, decision heuristics which consist of a *variable heuristics* and *phase saving*, as usual in state-of-the-art SAT solvers (Biere and Fröhlich 2015), can be used without further adjustments. This also includes *activity*-based schemes like EVSIDS (Eén and Sörensson 2004; Moskewicz et al. 2001).

(b) Similarly, the *lineral block distance* (Definition 4.43) with respect to an associated variable assignment stack can also be efficiently computed. Therefore, it can serve as a substitute for the *literal block distance* which is used in many *clause deletion heuristics*, see (Oh 2015), and dynamic restart heuristics, see (Audemard and Simon 2009; Biere and Fröhlich 2018). Therefore these heuristics can be used also in the context of CDXCL$^{\mathsf{lite}}$ without further modifications.

Since we only propagate literals to the clauses, backtracking to the assertion level of C might not lead to new literal propagations. Thus, we suggest the following.

Remark 5.18 (Backtracking Level)**.** Consider the setting of CDXCL$^{\mathsf{lite}}$ with the assignment stack $\mathcal{A} = (A_0, \ldots, A_d)$. In line 10 we use the *backtracking level* $d_{\mathtt{cl}} = \beta_{\mathcal{A}}(C)$ which is the assertion level of the $\mathcal{A}$-asserting clause C. Hence, C is a (non-trivial) unit clause under $A_{d_{\mathtt{cl}}}$. Observe that the corresponding implied lineral $\ell \in \mathbb{L}_n$ does not need to be assigning on this level, i.e., may not be a literal. In this case, the call to GCP$^{\mathsf{lite}}$ in line 11 does not yield any new literal propagations. To compensate for this, one can instead backtrack to the *literal assertion level* of C, defined as

$$d_{\mathtt{lit}} \;=\; \max\Big(\big\{\beta_{\mathcal{A}}(C)\big\} \cup \big\{\delta_{\mathcal{A}}(x_i) \mid x_i \in \mathrm{Vars}(\ell), \delta_{\mathcal{A}}(x_i) < \delta_{\mathcal{A}}(\ell)\big\}\Big).$$

By construction, we have $d_{\mathtt{cl}} \leq d_{\mathtt{lit}}$ and C is a non-trivial unit clause under $A_{d_{\mathtt{lit}}}$ whose implied lineral $\ell|_{A_{d_{\mathtt{lit}}}} \in \mathbb{X}_n$ is in fact a literal. Then GCP$^{\mathsf{lite}}$ can derive new literal propagations in line 11.

We suggest using the assertion level $d_{\mathtt{cl}}$, whenever $\mathrm{LBD}_{\mathcal{A}}(\ell|_{A_{d_{\mathtt{cl}}}}) \leq d_{\mathtt{lit}} - d_{\mathtt{cl}}$, i.e., when not all decisions after the $d_{\mathtt{cl}}$-th level are required to get the implied lineral assigning. Otherwise, backtrack to $d_{\mathtt{lit}}$, i.e., the assertion level of the implied lineral.

Based on Remark 5.7.a, we propose the following simplistic inprocessing method.

Remark 5.19 (Inprocessing)**.** In order to improve the propagation power of GCP$^{\mathsf{lite}}$ in line 6 of CDXCL$^{\mathsf{lite}}$, one can adjoin the literals in $\mathrm{ImpLits}(L_d)$ to L_d every now and then. Remark 5.9.d explains how corresponding reason clauses can be constructed.

To make the best of a new literal $\ell \in \mathrm{ImpLits}(L_d) \setminus L_d$ with reason clause $C \subseteq \mathbb{L}_n$, we proceed as follows: Compute the *assertion level* of C, i.e., the smallest $d_{\mathtt{cl}} \leq d$ such that C is a unit under $A_{d_{\mathtt{cl}}}$, and adjoin the implied lineral of C under $A_{d_{\mathtt{cl}}}$ to $L_{d_{\mathtt{cl}}}$. This ensures that, as long as the search does not backtrack over this decision level, the propagation GCP$^{\mathsf{lite}}$ is strengthened, and we can profit longer from a single execution of such an inprocessing

call. Moreover, we suggest backtracking immediately to the lowest level in which a newly implied lineral becomes a literal.

To see that CDXCL$^{\text{lite}}$ is a true combination of the speed of CDCL with the power of reasoning of CDXCL, we next introduce data structures to efficiently handle propagation with GCP$^{\text{lite}}$ and learning with LinTrailRes while offering constant-time backtracking in the context of CDXCL$^{\text{lite}}$.

5.3 Lazy Data Structures for Propagation

In this subsection, we focus on efficient implementations of GCP$^{\text{lite}}$ in the spirit of *watched literals* and *watch lists*, see (Eén and Sörensson 2004; Moskewicz et al. 2001). These are the key techniques that allow fast propagations and lazy backtracking in CNF-based conflict-driven SAT solving.

In the setting of GCP$^{\text{lite}}$, the main objectives for an assignment $A \subseteq \mathbb{X}_n$ are:

- Given a lineral $\ell \in \mathbb{L}_n$, check efficiently whether $\ell|_A$ is a non-zero literal.
- Given a clause $C \subseteq \mathbb{L}_n$, check efficiently whether $C|_A$ is a non-trivial unit clause.

At the same time, backtracking in the context of CDXCL$^{\text{lite}}$ should require no changes in the data structures.

For each objective, we propose a different data structure. Our suggestions follow the idea of watched literals: in each lineral or clause, we *watch* two variables; these are changed only when one of them newly assigned. Selecting the new watched variables appropriately allows performing the above checks in constant time. While this is rather straightforward for linerals, the selection must be done carefully for XNF clauses.

5.3.1 Watching Pairs for Linerals. For a variable assignment $A \subseteq \mathbb{L}_n$ and a lineral $\ell \in \mathbb{L}_n$, we want to check efficiently whether $\ell|_A$ is an assigning polynomial. This needs to be done for all assignments of an assignment stack, so we use a data structure that is updated incrementally, and allows to check $\ell|_A \in \mathbb{X}_n$ in constant time.

In (Soos, Nohl, et al. 2009), a watching scheme for XOR clauses is sketched. As XOR clauses correspond to unit XNF clauses and thus to linerals, their approach can also be used in our situation. For the sake of completeness, let us properly define and introduce this data structure but in our language of linerals.

Definition 5.20. Let $\ell \in \mathbb{L}_n \setminus \mathbb{X}_n$ be a lineral, and $A \subseteq \mathbb{X}_n$ be an assignment.

(a) A tuple $(i_1, i_2) \in \{1, \dots, n\}^2$ with

(WP_1) $i_1 \neq i_2$ and $x_{i_1}, x_{i_2} \in \operatorname{Vars}(\ell)$,

(WP_2) if x_{i_1} is assigned by A, then also x_{i_2} is assigned by A, and

(WP_3) if x_{i_2} is assigned by A, then $\ell|_A \in \mathbb{X}_n$,

is called an A-**watching pair** for ℓ. We also say that x_{i_1} and x_{i_2} are the **watched variables**, and ℓ is A-**watched by** (i_1, i_2). For simplicity we write $(i_1, i_2) \xrightarrow{A} \ell$.

(b) Let $\mathcal{A} = (A_0, \dots, A_d)$ be an assignment stack and $(i_1, i_2) \in \{1, \dots, n\}^2$ with $(i_1, i_2) \xrightarrow{A_i} \ell$ for all $i \in \{0, \dots, d\}$. Then (i_1, i_2) is called an $\mathcal{A}$-**watching pair** for ℓ, we say ℓ is $\mathcal{A}$-**watched by** (i_1, i_2), and we write $(i_1, i_2) \xrightarrow{\mathcal{A}} \ell$.

Given a watching pair $(i_1, i_2) \xrightarrow{\mathcal{A}} \ell$, we can determine whether ℓ is assigning under a variable assignment A_i of $\mathcal{A}$ by checking only the watched variable x_{i_2} as follows. Moreover, if the assignment stack is *fine* enough, we also know the variable of the implied literal. However, to determine the actual implied literal $\ell|_{A_i}$, a linear scan over $\operatorname{Supp}(\ell)$ is still required.

Proposition 5.21. Let $\ell \in \mathbb{L}_n$ be a lineral, $\mathcal{A} = (A_0, \dots, A_d)$ be a variable assignment stack, $(i_1, i_2) \xrightarrow{\mathcal{A}} \ell$ be a watching pair for ℓ, and $i \in \{0, \dots, d\}$.

(a) The watched variable x_{i_2} is assigned by A_i if and only if $\ell|_{A_i} \in \mathbb{X}_n$.

(b) If $\mathcal{A}$ is a stepwise assignment stack, x_{i_2} is assigned by A_i, and we have $i = \delta_{\mathcal{A}}(x_{i_2})$, then $\ell|_{A_i} \in \{x_{i_1}, x_{i_1} + 1\}$.

Proof. The implication in (a) is clear from (WP$_3$). For the converse, note that $\ell|_{A_i} \in \mathbb{X}_n$ implies that there is at most one variable in $\mathrm{Vars}(\ell)$ which is not assigned by A_i. Suppose for a contradiction that x_{i_2} is not assigned by A_i. Then (WP$_2$) implies that A_i does not assign x_{i_1} as well. Hence we get $i_1 = i_2$ which contradicts (WP$_1$). Consequently, x_{i_2} must be assigned by A_i.

For (b), observe that the watched variables x_{i_1} and x_{i_2} cannot have the same implication level, since $\mathcal{A}$ is a stepwise assignment stack. Now $i = \delta_{\mathcal{A}}(x_{i_2})$ and (WP$_2$) imply that x_{i_1} is not assigned by A_i, and the claim follows from (WP$_3$). □

In order to put this proposition into use, we need to maintain a watching pair, i.e., properties (WP$_1$) – (WP$_3$), during CDXCL$^{\text{lite}}$. For this, we have the next algorithm.

Algorithm 7: UpdWatchPair – Update Watching Pairs

Input : An A-watching pair (i_1, i_2) for $\ell \in \mathbb{L}_n$, an assignment A' with $A \subseteq A'$.
Output: An (A, A')-watching pair (i_1, i_2) for ℓ.

1 **if** x_{i_1} *is assigned by* A' **then**
2 | **if** *there is* $x_j \in \mathrm{Vars}(\ell) \setminus \{x_{i_2}\}$ *not assigned by* A' **then** let $i_1 = j$.
3 **if** x_{i_2} *is assigned by* A' **then**
4 | **if** *there is* $x_j \in \mathrm{Vars}(\ell) \setminus \{x_{i_1}\}$ *not assigned by* A' **then** let $i_2 = j$.
5 **if** x_{i_1} *is assigned by* A' **then** swap i_1 and i_2.
6 **return** (i_1, i_2).

Proposition 5.22. Let (A, A') be a variable assignment stack and let $\ell \in \mathbb{L}_n$ with a watching pair $(i_1, i_2) \xrightarrow{A} \ell$. Then UpdWatchPair is an algorithm which returns an (A, A')-watching pair for ℓ.

Proof. It suffices to show the correctness of the procedure. By construction, we have $j \notin \{i_1, i_2\}$ in lines 2 and 4. Thus, in line 6 we get $i_1 \neq i_2$. This shows (WP$_1$) for A'. Line 5 ensures that the output satisfies property (WP$_2$) for A'. Suppose that $\ell|_{A'} \notin \mathbb{X}_n$. Then, at least two variables in $\mathrm{Vars}(\ell)$ are not assigned by A'. These indices can be picked for i_1 and i_2 respectively in lines 2 and 4. In particular, x_{i_2} is not assigned by A', and we have (WP$_3$) for A'. Therefore, (i_1, i_2) is an A' watching pair for ℓ in line 6.

Moreover, updates to the watched variables only happen if the updated watched variables are unassigned by A'. As $A \subseteq A'$, the new variables are also unassigned by A. This shows that (WP$_2$) and (WP$_3$) also hold for A, i.e., (i_1, i_2) is also an A-watching pair. □

Remark 5.23. Let $\mathcal{A} = (A_0, \ldots, A_d)$ be a variable assignment stack and let (i_1, i_2) be an $(A_0, \ldots, A_{d-1})$-watching pair for some lineral $\ell \in \mathbb{L}_n$. Then the output of $\mathrm{UpdWatchPair}_\ell((i_1, i_2), A_d)$ is an $\mathcal{A}$-watching pair for ℓ. With Proposition 5.21.a we can quickly check whether $\ell|_{A_d} \in \mathbb{X}_n$. Additionally, whenever we backtrack during CDXCL$^{\text{lite}}$ to a decision level $d' < d$, we can use (i_1, i_2) as an $(A_0, \ldots, A_{d'})$-watching pair for ℓ. Thus backtracking does not necessitate any changes in the data structures.

To obtain an efficient implementation, consider the following remarks.

Remark 5.24 (Sparse Representation of Linerals)**.** To allow fast manipulation of linerals, we use the following *sparse representation*: A lineral $\ell \in \mathbb{L}_n$ is stored in memory as a *sorted vector* of integers representing the set $\{i \in \{1, \ldots, n\} \mid x_i \in \mathrm{Vars}(\ell)\}$ along with a *boolean value* indicating whether $1 \in \mathrm{Supp}(\ell)$.

As this vector presents the variables of ℓ in lexicographic order, the leading term $\mathrm{LT}_{\mathsf{Lex}}(\ell)$ can be determined in constant time as its first element. Therefore, whenever a term ordering is required, we choose $\sigma = \mathsf{Lex}$. Also, with this representation, the sum of two linerals $\ell_1, \ell_2 \in \mathbb{L}_n$ can essentially be computed by a linear scan over their sorted vectors, i.e., can be determined in $O\big(\#\,\mathrm{Supp}(\ell_1) + \#\,\mathrm{Supp}(\ell_2)\big)$ time and space.

Unlike CNF-based SAT solving, we sometimes want to set *expiration dates* to specific pieces of information that is cached, or costly to re-compute. To realize this, we suggest a decision-level counting scheme as follows.

Remark 5.25 (Decision Level Counting)**.** During a run of CDXCL$^{\mathsf{lite}}$, let us count for each decision level d the number of times the algorithm backtracks *from* and *over* level d, and store it as $\mathcal{T}(d) \in \mathbb{N}$, the **dl-count** of d. Now, whenever we want to remember that some information $\mathcal{I}$ is valid until the solver backtracks to a decision level lower than d', we store the **dl-count tuple** (d', c') with $c' = \mathcal{T}(d')$ along with $\mathcal{I}$. Remember that $\mathcal{T}$ changes with each new decision level and backtracking of CDXCL$^{\mathsf{lite}}$. But as long as $\mathcal{T}(d') = c'$, we know that the information $\mathcal{I}$ is still valid. Only if $\mathcal{T}(d') > c'$, the information $\mathcal{I}$ is considered expired and thus invalid.

Remark 5.26 (Implementation Details)**.**

(a) By storing variable assignments $A \subseteq \mathbb{X}_n$ as n-tuples of values in $\{0, 1, \mathfrak{u}\}$, where $\mathfrak{u}$ encodes *unassigned* values, we can check if a variable x_i is assigned by A in constant time. In this way, UpdWatchPair can be implemented with a running time in $O\big(\#\,\mathrm{Vars}(\ell)\big)$.

(b) To select the variables in lines 2 and 4, we suggest proceeding in a *circular* manner: Pick the first unassigned variable in Lex-order after the watched variable, which is to be updated. If there is none, continue from the smallest term of ℓ. This mimics the flow of an (amortized) optimal update scheme for watched literals of CNF clauses, see (Gent 2013).

(c) Note that UpdWatchPair changes an input watching pair only if one of the watched variables is newly assigned by A'. Thus, when a new assignment $\ell \in \mathbb{X}_n$ is propagated in GCP$^{\mathsf{lite}}$, it suffices to update only those watching pairs where $\mathrm{LT}_\sigma(\ell)$ is one of the watched variables. Similar to (Eén and Sörensson 2004), this allows us to store all the watching pairs in so-called *watch lists*, such that only the watching pairs that need an update are visited. For each variable x_i we therefore track a *watch list* L_{x_i} containing (pointers to) the all watching pairs which watch the variable x_i. The watch lists can be kept up to date during propagation by moving entries between the lists.

As new linerals may be added to L during GCP$^{\mathsf{lite}}$, new watching pairs need to be created and added to the respective lists. When the solver backtracks, the corresponding entries need to be removed.

(d) Removing the entries of the watch lists in the setting of part (c) is not desirable, as this loads many watch lists into working memory only to remove certain elements from them. This significantly impacts the speed of backtracking, which we would like to be of constant time. So we suggest giving each watch list entry an *expiration time* using a *dl-count tuple*, as described in Remark 5.25. Now, whenever a watch list is iterated and an entry has *expired*, we can simply remove it. Otherwise, we proceed as usual to visit the corresponding lineral and update the watched variables. Altogether, this enables lazy housekeeping of the watch lists.

Finally, note that GCP$^{\mathsf{lite}}_{\mathsf{GJE}}$ can also be efficiently implemented using ideas from (C.-S. Han and Jiang 2012; Laitinen et al. 2012c; Soos, Gocht, et al. 2020).

5.3.2 Watching Structures for XNF Clauses. This section presents a data structure for XNF clauses $C \subseteq \mathbb{L}_n$ and assignment stacks $\mathcal{A} = (A_0, \ldots, A_d)$ which can efficiently check whether $C|_{A_i}$ is a non-trivial unit clause. The data structure requires only incremental updates and does not need to be changed upon backtracking in the context of CDXCL$^{\mathsf{lite}}$. It is based around the same core idea of *watched literal* schemes. In particular, it watches

two distinct linerals ℓ_{j_1}, ℓ_{j_2} of C by selecting one watched variable in each lineral. In order to not miss the clause becoming a unit, these variables must not be shared by ℓ_{j_1} and ℓ_{j_2}, i.e., may not be in $\mathrm{Vars}(\ell_{j_1}) \cap \mathrm{Vars}(\ell_{j_2})$. This necessitates changing the representation of the clause C itself if no unassigned unshared variables are left in the watched linerals. While this has a negative impact on the performance of the update step, it directly yields a reason clause decomposition of V_C. Eventually, this leads to an efficient implementation of LinTrailRes as described in the next section.

Let us proceed as follows: First, we formerly introduce our watching scheme and prove that it serves the claimed purpose. Then, we show how it can be initialized and kept up to date before we close with some implementation remarks and an example.

Definition 5.27. Let $C \subseteq \mathbb{L}_n$ be an XNF clause which is not a unit clause, let $A \subseteq \mathbb{X}_n$ be an assignment, and let $\mathcal{A} = (A_0, \dots, A_d)$ be a variable assignment stack.

(a) A tuple $W = \big((\ell_1, \dots, \ell_k),\ (i_1, j_1),\ (i_2, j_2)\big)$ with $k \in \mathbb{N}$, $\ell_1, \dots, \ell_k \in \mathbb{L}_n$, $i_1, i_2 \in \{1, \dots, n\}$, $j_1, j_2 \in \{1, \dots, k\}$, and

(ws_1) $V_C = \langle \ell_1, \dots, \ell_k \rangle_{\mathbb{F}_2}$,

(ws_2) $j_1 \neq j_2$, $x_{i_1} \in \mathrm{Vars}(\ell_{j_1}) \setminus \mathrm{Vars}(\ell_{j_2})$, $x_{i_2} \in \mathrm{Vars}(\ell_{j_2}) \setminus \mathrm{Vars}(\ell_{j_1})$,

(ws_3) if x_{i_1} is assigned by A, then also x_{i_2} is assigned by A,

(ws_4) if x_{i_2} is assigned by A, then $\ell_{j_2}|_A \in \mathbb{F}_2$,

(ws_5) if $\ell_{j_2}|_A = 0$, then $\ell_j|_A = 0$ for all $j \in \{1, \dots, k\} \setminus \{j_1\}$,

is called an A-**watching structure** for C. We also say that x_{i_1}, x_{i_2} are the **watched variables**, ℓ_{j_1}, ℓ_{j_2} the **watched linerals**, and C is A-**watched** by W. For simplicity, we also write $W \xrightarrow{A} C$.

(b) Let W be an A_i-watching structure for C, for all $i \in \{0, \dots, d\}$. Then the tuple W is called an $\mathcal{A}$-**watching structure** for C, we say C is $\mathcal{A}$-**watched by** W, and we write $W \xrightarrow{\mathcal{A}} C$.

According to this definition, $\ell_1, \dots, \ell_k$ need not form a basis of V_C, but we have $C \equiv \{\ell_1 + 1, \dots, \ell_k + 1\}$ by Proposition 2.12. As desired, watching structures can be used to efficiently check whether $C|_A$ is a non-trivial unit clause as follows.

Proposition 5.28. Let $A \subseteq \mathbb{X}_n$ be a variable assignment, let $C \subseteq \mathbb{L}_n$ be an XNF clause, and let

$$\big((\ell_1, \dots, \ell_k),\ (i_1, j_1),\ (i_2, j_2)\big) \xrightarrow{A} C$$

be a watching structure for C.

(a) If $\ell_{j_1}|_A = 1$ or $\ell_{j_2}|_A = 1$, then C is satisfied by A.

(b) The following conditions are equivalent.

(1) $C|_A$ is a non-trivial unit clause, i.e., $C|_A \equiv \{\ell\}$ for some $\ell \in \mathbb{L}_n \setminus \{0\}$.

(2) The watched linerals satisfy $\ell_{j_1}|_A \neq 1$ and $\ell_{j_2}|_A = 0$.

(3) The watched variable x_{i_2} is assigned by A, $\ell_{j_1}|_A \neq 1$, and $\ell_{j_2}|_A \neq 1$.

In this case C is a reason clause under A with reason lineral $\ell_{j_1} + 1$. Moreover, the reason clause decomposition of V_C is given by

$$V_C \;=\; \langle\, \ell_j \mid j \in \{1, \dots, k\} \setminus \{j_1\} \,\rangle_{\mathbb{F}_2} \oplus \langle \ell_{j_1} \rangle_{\mathbb{F}_2}.$$

Proof. Part (a) is a consequence of (ws_1) and Proposition 2.12.b. For the implication (1)⇒(2) in (b), let $C|_A$ be a non-trivial unit clause. In particular this means that A is a valid assignment and there is $\ell \in \mathbb{L}_n \setminus \{0\}$ with $C|_A \equiv \{(\ell_1 + 1)|_A, \dots, (\ell_k + 1)|_A\} \equiv \{\ell\}$. This implies $V_{C|_A} = \langle \ell_1|_A, \dots, \ell_k|_A \rangle_{\mathbb{F}_2} = \langle \ell + 1 \rangle_{\mathbb{F}_2}$, i.e., for $j \in \{1, \dots, k\}$, we have $\ell_j|_A = \ell + 1$ or $\ell_j|_A = 0$. Assume for a contradiction that x_{i_2} is not assigned by A. Then (ws_3) implies that A does not assign x_{i_1} as well. Consequently, we get $\ell_{j_1}|_A \neq 0$ and $\ell_{j_2}|_A \neq 0$. From the above we obtain $\ell_{j_1}|_A = \ell_{j_2}|_A = \ell + 1$. Hence we get

$$x_{i_1}, x_{i_2} \in \operatorname{Vars}(\ell_{j_1}|_A) \cap \operatorname{Vars}(\ell_{j_2}|_A) \subseteq \operatorname{Vars}(\ell_{j_1}) \cap \operatorname{Vars}(\ell_{j_2})$$

in contradiction to (ws_2). The watched variable x_{i_2} must therefore be assigned by A, and with (ws_4) we get $\ell_{j_2}|_A = 0$. By (ws_5) this shows $\ell_j|_A = 0$ for all $j \in \{1, \dots, k\} \setminus \{j_1\}$, and the above gives $\ell_{j_1}|_A = \ell + 1 \neq 1$. This shows (2).

Conversely, assume that $\ell_{j_2}|_A = 0$ and $\ell_{j_1}|_A \neq 1$. Then (ws_5) implies $\ell_j|_A = 0$ for all $j \in \{1, \dots, k\} \setminus \{j_1\}$, i.e., $C|_A \equiv \{(\ell_{j_1} + 1)|_A\}$ is a non-trivial unit clause.

Thus we have (1)⇔(2), and the equivalence of (2) and (3) follows from (ws_4). □

Remark 5.29. Consider the situation of Proposition 5.28. If the watched variable x_{i_2} is assigned by the variable assignment $A \subseteq \mathbb{X}_n$, then C is a unit clause under A. In particular, we have

$$C|_A \equiv \begin{cases} \{(\ell_{j_1} + 1)|_A\} & \text{if } \ell_{j_2}|_A = 0 \\ \top & \text{otherwise.} \end{cases}$$

So the value of $\ell_{j_2}|_A$ tells us whether C is satisfied by A or a unit clause under A.

Additionally, if C is an asserting clause, then the decision level of the second watched variable tells us the assertion level of C.

Proposition 5.30. Let $\mathcal{A}$ be a variable assignment stack, let $C \subseteq \mathbb{L}_n$ be an $\mathcal{A}$-asserting XNF clause, and let $\big((\ell_1, \dots, \ell_k),\ (i_1, j_1),\ (i_2, j_2)\big) \xrightarrow{\mathcal{A}} C$ be an $\mathcal{A}$-watching structure for C. Then we have $\beta_{\mathcal{A}}(C) = \delta_{\mathcal{A}}(x_{i_2})$.

Proof. By (ws_5) we have $\delta_{\mathcal{A}}(\ell_{j_2}) \geq \delta_{\mathcal{A}}(\ell_j)$ for all $j \in \{1, \dots, k\} \setminus \{j_1\}$. With $\mathcal{A} = (A_0, \dots, A_d)$ the reason clause decomposition in Proposition 5.28.b yields

$$\begin{aligned} \beta_{\mathcal{A}}(C) &= \min\big\{i \in \{0, \dots, d-1\} \mid C|_{A_i} \text{ is a unit clause}\big\} \\ &= \min\big\{i \in \{0, \dots, d-1\} \mid V_C \cap \langle A_i \rangle_{\mathbb{F}_2} = V_C \cap \langle A_{d-1} \rangle_{\mathbb{F}_2}\big\} \\ &= \max\big\{\delta_{\mathcal{A}}(\ell_j) \mid j \in \{1, \dots, k\} \setminus \{j_1\}\big\} = \delta_{\mathcal{A}}(\ell_{j_2}). \end{aligned}$$

Finally, (ws_4) implies $\delta_{\mathcal{A}}(\ell_{j_2}) = \delta_{\mathcal{A}}(x_{j_2})$, and the claim follows. □

Next, we propose *initialization* and *update* procedures.

Remark 5.31 (Initialization)**.** Let $C \subseteq \mathbb{L}_n$ be an XNF clause, which is not a unit clause. Compute a basis $\{\ell_1, \dots, \ell_k\}$ of V_C. As C is no unit clause, we have $k \geq 2$. If $\operatorname{Vars}(\ell_1) \setminus \operatorname{Vars}(\ell_2) = \emptyset$ or $\operatorname{Vars}(\ell_2) \setminus \operatorname{Vars}(\ell_1) = \emptyset$, then replace ℓ_1 with $\ell_1 + \ell_2$. Then there are $x_{i_1} \in \operatorname{Vars}(\ell_1) \setminus \operatorname{Vars}(\ell_2)$ and $x_{i_2} \in \operatorname{Vars}(\ell_2) \setminus \operatorname{Vars}(\ell_1)$, and

$$\big((\ell_1, \dots, \ell_k),\ (i_1, 1),\ (i_2, 2)\big) \xrightarrow{\emptyset} C$$

is an $\emptyset$-watching structure for C.

Proposition 5.32. Let $C \subseteq \mathbb{L}_n$ be an XNF clause, let $\mathcal{A} = (A_0, \dots, A_d)$ be a stepwise variable assignment stack, and let $W = \big((\ell_1, \dots, \ell_k),\ (i_1, j_1),\ (i_2, j_2)\big)$ be an $(A_0, \dots, A_{d-1})$-watching structure for C. Then UpdWatchStruct is an algorithm which returns an $\mathcal{A}$-watching structure for C.

Algorithm 8: UpdWatchStruct – Update Watching Structures

Input : A stepwise variable assignment stack $\mathcal{A} = (A_0, \dots, A_d)$, an $(A_0, \dots, A_{d-1})$-watching structure $\big((\ell_1, \dots, \ell_k),\ (i_1, j_1),\ (i_2, j_2)\big)$ for a clause $C \subseteq \mathbb{L}_n$.

Output : An $\mathcal{A}$-watching structure $\big((\ell_1, \dots, \ell_k),\ (i_1, j_1),\ (i_2, j_2)\big)$ for C.

1 Write $A_d = A_{d-1} \cup \{x_l + a\}$ for some $x_l + a \in \mathbb{X}_n$.
2 **if** $l \notin \{i_1, i_2\}$ **then return** $\big((\ell_1, \dots, \ell_k),\ (i_1, j_1),\ (i_2, j_2)\big)$.
3 Let $\{s, t\} = \{1, 2\}$ with $i_s = l$.
4 **if** *there is* $x_i \in \mathrm{Vars}(\ell_{j_s})$ *not assigned by* A_d **then** update watched variable
5 Let $i_s = i$.
6 **if** $x_{i_s} \in \mathrm{Vars}(\ell_{j_t})$ **then** replace ℓ_{j_t} with $\ell_{j_t} + \ell_{j_s}$.
7 **else if** $\ell_{j_s}|_{A_d} = 0$ **then** update watched lineral
8 **for** $j \in \{1, \dots, k\} \setminus \{j_1, j_2\}$ **do**
9 **if** $x_{i_t} \in \mathrm{Vars}(\ell_j)$ **then** replace ℓ_j with $\ell_j + \ell_{j_t}$.
10 **if** $\ell_j|_{A_d} \neq 0$ **then** found new watched lineral
11 Let $x_i \in \mathrm{Vars}(\ell_j)$ with $\begin{cases} \delta_{\mathcal{A}}(x_i) \text{ maximal} & \text{if } \ell_j|_{A_d} = 1, \\ x_i \text{ not assigned by } A_d & \text{otherwise.} \end{cases}$
12 **if** $x_i \in \mathrm{Vars}(\ell_{j_t})$ **then** replace ℓ_{j_t} with $\ell_j + \ell_{j_t}$.
13 Let $(i_s, j_s) = (i, j)$, and **continue** in line 14.
14 **if** x_{i_1} *is assigned by* A_d *and* x_{i_2} *is not assigned by* A_d **then**
15 Swap (i_1, j_1) and (i_2, j_2).
16 **return** $\big((\ell_1, \dots, \ell_k),\ (i_1, j_1),\ (i_2, j_2)\big)$.

Proof. It suffices to show correctness if the procedure terminates in line 16. Because of (ws_2) and the fact that $\mathcal{A}$ is a stepwise assignment stack, the procedure (ignoring the swap in line 15) only updates at most one of the two watched variables, namely x_{i_s} along with its watched lineral ℓ_{i_s}. Let us now show (ws_1) – (ws_5).

Whenever a lineral ℓ_j is replaced (lines 6, 9, and 12), it arises as the sum of ℓ_j with another lineral from $\{\ell_1, \dots, \ell_k\} \setminus \{\ell_j\}$. Hence, the output satisfies (ws_1).

To see the other properties, let us follow the flow of the algorithm: first, in lines 4-6, we try to find a new unassigned variable inside of ℓ_{j_s} to watch. If this fails, we have $\ell_{j_s}|_{A_d} \in \mathbb{F}_2$ after line 6. Now either $\ell_{j_s}|_{A_d} = 1$, i.e., C is satisfied by A_d, or $\ell_{j_s}|_{A_d} = 0$. In the latter case, a new watched lineral along with a new watched variable is searched for, see lines 7-13. Only if there is an unwatched lineral ℓ_j which evaluates to 1 under A_d or has an unassigned variable (line 11), the watched lineral and variable are updated. If no update occurs, then $\ell_j|_{A_d} = 0$, for all $j \in \{1, \dots, k\} \setminus \{j_t\}$. In view of lines 14-15 this shows (ws_5). Moreover, after line 13, x_{i_s} is only assigned by A_d if $\ell_{j_s}|_{A_d} \in \mathbb{F}_2$. This shows ($\mathrm{ws}_4$). Since W is a watching structure for C, we also have $x_{i_t} \in \mathrm{Vars}(\ell_{j_s} + \ell_{j_t})$ and $\mathrm{Vars}(\ell_{j_s}) \cap \mathrm{Vars}(\ell_{j_t}) \cap \mathrm{Vars}(\ell_{j_s} + \ell_{j_t}) = \emptyset$. Hence line 6 ensures ($\mathrm{ws}_2$). Similarly, line 9 ensures $x_{i_t} \notin \mathrm{Vars}(\ell_j)$, and $x_i \notin \mathrm{Vars}(\ell_{j_t})$ by line 12. Altogether, (ws_2) also holds after line 13. Property (ws_3) is enforced by lines 14-15. This shows that the output W in line 16 is indeed an A_d-watching structure for C.

Finally, for all $i \in \{0, \dots, d-1\}$, a watched variable can only change to a variable which is not assigned by A_i or whose corresponding lineral evaluates to 1 (see line 11). This means that properties (ws_3)-(ws_5) are satisfied for the assignment A_i as well. Thus, W is also an A_i-watching structure for C, and therefore an $\mathcal{A}$-watching structure for C as desired. □

Let us conclude this section with comments on extensions, optimizations, and some caveats for UpdWatchStruct.

Remark 5.33. The check of $\ell_j|_{A_d} \neq 0$ in line 10 of UpdWatchStruct requires a linear scan of $\mathrm{Supp}(\ell_j)$. This is rather expensive if $\#\,\mathrm{Supp}(\ell_j)$ is large, as this condition may be checked often during repeated calls to UpdWatchStruct. To counteract this, we *cache* this information using the decision level counting technique of Remark 5.25. To be precise, whenever a lineral $\ell \in \mathbb{L}_n$ is evaluated to 0 (or 1) in level d', we store the corresponding *dl-count tuple* along with ℓ. Now $\ell|_{A_d} = \ell|_{A_{d'}}$ for all $d \geq d'$, and we only need to recompute $\ell|_{A_d}$ once the *expiration time* set by the *dl-count tuple* has passed, i.e., if we backtrack to a decision level lower than d'. Additionally, when we choose the value of d' minimal with $\ell|_{A_{d'}} = 0$ (or 1), then we have $\delta_{\mathcal{A}}(\ell) = d'$.

Now that this check can be performed efficiently, we can also skip the linerals with $\ell_j|_{A_d} = 0$ in line 9 right away to avoid unnecessary replacements.

Remark 5.34 (Disjoint Watching Structures)**.** Let $\mathcal{A}$ be a stepwise variable assignment stack and let $W = \big((\ell_1, \ldots, \ell_k), (i_1, j_1), (i_2, j_2)\big)$ be an $\mathcal{A}$-watching structure for an XNF clause C. If, for all $\ell_i, \ell_j \in \{\ell_1, \ldots, \ell_k\}$ we have $\mathrm{Vars}(\ell_i) \cap \mathrm{Vars}(\ell_j) = \emptyset$, then we call W a **disjoint watching structure**. The benefit of disjoint watching structures is that computationally expensive parts of UpdWatchStruct become irrelevant because (ws_2) reduces to $j_1 \neq j_2$. In particular, all the *replacements* in lines 6, 9, and 12 can be skipped entirely, and the linerals $\ell_1, \ldots, \ell_k$ are not changed at all. Consequently, the output of $\mathrm{UpdWatchStruct}_{\mathcal{A}}(W)$ is also a disjoint watching structure.

For instance, all watching structures for CNF clauses are disjoint. A more general approach to optimizing UpdWatchStruct is explained in the next remark.

Remark 5.35 (Implementation Details)**.**

(a) To efficiently implement the replacements in lines 6, 9, and 12, we store the watched linerals not explicitly as ℓ_{j_1} and ℓ_{j_2}. Instead, we track linerals ℓ'_{j_1}, ℓ'_{j_2}, and ℓ_{shared} with $\ell_{j_s} = \ell'_{j_s} + \ell_{\mathsf{shared}}$ and $\mathrm{Vars}(\ell_{\mathsf{shared}}) = \mathrm{Vars}(\ell_{j_1}) \cap \mathrm{Vars}(\ell_{j_2})$. Then ℓ'_{j_1} and ℓ'_{j_2} have no common variables.

Consequently, in line 4 we can first search for a new unassigned variable in ℓ'_{j_s}. If we are successful, line 6 can be skipped entirely. Otherwise, we check if ℓ_{shared} contains an unassigned variable and perform the replacement of line 6. However, because of our decomposition, this boils down to *swapping* the linerals ℓ'_{j_s} and ℓ_{shared}. This can even be realized as a constant time operation. Similar adaptions can be done for lines 9 and 12. The only apparent disadvantage is that whenever one of the watched linerals changes (line 10), the above decompositions need to be recomputed.

(b) Since we store linerals as sorted vectors (see Remark 5.26), the *shared* part of two linerals $\ell_1, \ell_2 \in \mathbb{L}_n$ can be computed with a linear scan in time $O(\#\,\mathrm{Vars}(\ell_1) + \#\,\mathrm{Vars}(\ell_2)) = O(n)$.

(c) To choose a new watched variable in lines 4 and 11, we suggest searching circularly, i.e., starting from the currently watched variable we select the next unassigned variable in Lex-order. If there is none, we continue with the leading term of the lineral. This mimics the flow of an (amortized) optimal update scheme for watched literals of CNF clauses, see (Gent 2013).

(d) Analogous to Remark 5.26.c, an implementation of $\mathrm{GCP}^{\mathrm{lite}}$ should determine the clauses which need to be checked using *watch lists* for each variable. These store pointers to all watching structures where the respective variable is watched.

Altogether, $\mathrm{GCP}^{\mathrm{lite}}$ keeps track of two different types of watch lists; one for the *watching pairs* responsible for the linerals in the set $L \subseteq \mathbb{L}_n$, and one for *watching structures* responsible for the clauses $C \in F$. During backtracking in the context of $\mathrm{CDXCL}^{\mathrm{lite}}$, the watch lists, the watching pairs, and the watching structures need not be updated. It is sufficient to remove the variable assignments from A_d.

To illustrate our update scheme for watching structures, consider the following example.

Example 5.36. Let $\mathcal{A} = (A_0, A_1, A_2, A_3)$ be a stepwise assignment stack corresponding to the chain $\emptyset \subset \{x_1\} \subset \{x_1, x_2\} \subset \{x_1, x_2, x_4\}$ and consider the XNF clause

$$C = \{x_2 + x_3,\ x_1 + 1,\ x_1 + x_2 + x_4 + 1,\ x_4 + 1\} \subseteq \mathbb{L}_4.$$

Let us show how an $\mathcal{A}$-watching structure for C can be derived using UpdWatchStruct.

To ease the presentation, we underline the first watched variable and highlight the second watched variable with a dashed line.

(0) A basis of V_C is given by $1 + C$, and by Remark 5.31 we have

$$\Big(\big(\underline{x_2} + x_3 + 1, \underline{x_1},\ x_1 + x_2 + x_4,\ x_4\big),\ (2,1),\ (1,2)\Big) \xrightarrow{A_0} C.$$

(1) To get an (A_0, A_1)-watching structure, we can use UpdWatchStruct. As the second watched variable x_1 is assigned by A_1, we try to find another unassigned variable in the second watched lineral $\ell_2 = x_1$ (lines 4 to 6). As there are no such variables and we have $\ell_2|_{A_1} = 0$, we need to consider watching a different lineral, e.g., $\ell_3 = x_1 + x_2 + x_4$ (line 8). Since the watched variable x_2 is contained in $\mathrm{Vars}(\ell_3)$, we replace ℓ_3 by $\ell_3 + \ell_1 = x_1 + x_3 + x_4 + 1$ (line 9). Now $\ell_3|_{A_1} = x_3 + x_4 + 1$, and we can select x_3 as the new second watched variable (line 11). With $x_3 \in \mathrm{Vars}(\ell_1)$, line 12 requires us to replace ℓ_1 by $\ell_1 + \ell_3 = x_1 + x_2 + x_4$ before we get

$$\Big(\big(x_1 + \underline{x_2} + x_4,\ x_1,\ x_1 + \underline{x_3} + x_4 + 1,\ x_4\big),\ (2,1),\ (3,3)\Big) \xrightarrow{(A_0, A_1)} C.$$

(2) Note that $A_2 = \{x_1, x_2\}$ newly assigns the variable x_2, i.e., the first watched variable. Now UpdWatchStruct can select $x_4 \in \ell_1$ as the new first watched variable (line 4). As we have $x_4 \in \mathrm{Vars}(\ell_3)$, we replace ℓ_3 by $\ell_3 + \ell_1 = x_2 + x_3 + 1$ in line 6. Then we get

$$\Big(\big(x_1 + x_2 + \underline{x_4},\ x_1,\ x_2 + \underline{x_3} + 1,\ x_4\big),\ (4,1),\ (3,3)\Big) \xrightarrow{(A_0, A_1, A_2)} C.$$

(3) Finally, $A_3 = \{x_1, x_2, x_4\}$ assigns the first watched variable x_4. Now that there are no more unassigned variables in ℓ_1 and due to $\ell_1|_{A_3} = 0$, we try to choose a new lineral to watch in lines 8 to 13. Both the linerals ℓ_2 and ℓ_4 are checked, we have $\ell_2|_{A_3} = \ell_4|_{A_3} = 0$. So, the algorithm does not change the watched linerals or variables there. But it performs a swap of them in line 15, and we get

$$\Big(\big(x_1 + x_2 + \underline{x_4},\ x_1,\ x_2 + \underline{x_3} + 1,\ x_4\big),\ (3,3),\ (4,1)\Big) \xrightarrow{\mathcal{A}} C.$$

Since the second watched variable x_4 is assigned by $A_3 = \{x_1, x_2, x_4\}$, Remark 5.29 shows that C is a unit clause under A_3 with $C|_{A_d} \equiv \{(\ell_3 + 1)|_{A_3}\} \equiv \{x_3\}$. Moreover, Proposition 5.28.b yields the following reason clause decomposition

$$V_C = \langle \ell_1, \ell_2, \ell_4 \rangle_{\mathbb{F}_2} \oplus \langle \ell_3 \rangle_{\mathbb{F}_2} = \langle x_1 + x_2 + x_4,\ x_1,\ x_4 \rangle_{\mathbb{F}_2} \oplus \langle x_2 + x_3 + 1 \rangle_{\mathbb{F}_2}.$$

Using the modifications of Remark 5.33, we have *dl-count tuples* which tell us that $\delta_{\mathcal{A}}(\ell_2) = 1$ and $\delta_{\mathcal{A}}(\ell_1) = \delta_{\mathcal{A}}(\ell_4) = 3$. This information will prove useful in the next section.

In the next subsection we show how watching structures give rise to an efficient implementation of LinTrailRes.

5.4 Lazy Data Structures for Clause Learning

Let $\mathcal{L} = (L_0, \ldots, L_d)$ be a lite conflict solver state for an XNF formula F along with the associated variable assignment stack $\mathcal{A}$ and the associated stepwise variable assignment stack $\mathcal{A}' = (A'_0, \ldots, A'_t)$. Recall that we denote the i-th assignment subspace $\langle A'_i \rangle_{\mathbb{F}_2}$ by $\mathcal{A}'_i$, see Definition 4.23.

Let us first identify the key objectives for an efficient implementation of LinTrailRes. During an execution of LinTrailRes, we keep track of a generating system of a subspace $V \subseteq \mathbb{L}_n$ corresponding to a conflict clause C.

This subspace is changed until the associated clause C is $\mathcal{A}$-asserting. Whenever V is updated, we decompose it in the style of a *reason clause decomposition*, i.e., as $V = (V \cap \mathcal{A}'_i) \oplus \langle \ell_C + 1 \rangle_{\mathbb{F}_2}$ for some $i \leq t$. Then V is replaced by

$$V = \big(V \cap \mathcal{A}'_i\big) + \big(V_R \cap \mathcal{A}'_i\big) + \langle \ell_C + \ell_R + 1 \rangle_{\mathbb{F}_2},$$

where $R \in F$ is a reason clause with reason lineral ℓ_R from the d-th lite lineral trail. Observe that if an $\mathcal{A}'$-watching structure is known for R, then also a reason lineral ℓ_R and generators of $V_R \cap \mathcal{A}'_i$ are known by Proposition 5.28.b. In the next update step, this subspace V is intersected again, however, with a smaller set $\mathcal{A}'_j$ with $j < i$.

Consequently, the whole clause learning procedure consists of essentially two tasks: *Intersecting* V with ever-decreasing subspaces $\mathcal{A}'_i$ over and over again, and *extending* a generating set of this subspace with known linerals. Hence, our data structure should realize both these operations in a computationally efficient way. For this, we introduce the following concept of *filtrations*.

Definition 5.37. Let $\mathcal{A} = (A_0, \dots, A_d)$ be a variable assignment stack and $C \subseteq \mathbb{L}_n$ be an XNF reason clause under A_d.

(a) A tuple $\mathcal{B} = (B_0, \dots, B_d, B_\infty)$ of pairwise disjoint subsets of $\mathbb{L}_n \setminus \{0\}$, where

(1) $B_0 \cup \dots \cup B_d \cup B_\infty$ is a generating system of V_C,

(2) $V_C \cap \mathcal{A}_d = \langle B_0 \cup \dots \cup B_d \rangle_{\mathbb{F}_2}$ and $\#B_\infty \leq 1$,

(3) $B_i \subseteq \mathcal{A}_i$, for each $i \in \{0, \dots, d\}$, and

(4) $B_i \cap \mathcal{A}_{i-1} = \emptyset$, for each $i \in \{1, \dots, d\}$,

is called an $\mathcal{A}$**-filtration** of C. The number

$$\lambda(\mathcal{B}) = \max\big\{\, i \in \{0, \dots, d, \infty\} \mid B_i \neq \emptyset \,\big\}$$

is called the **leading level** of $\mathcal{B}$.

(b) An $\mathcal{A}$-filtration $\mathcal{B} = (B_0, \dots, B_d, B_\infty)$ of C with $\#B_{\lambda(\mathcal{B})} = 1$ is called a **watchable** $\mathcal{A}$**-filtration**.

(c) An $\mathcal{A}$-filtration $\mathcal{B} = (B_0, \dots, B_d, B_\infty)$ with $\langle B_0 \cup \dots \cup B_i \rangle_{\mathbb{F}_2} = V_C \cap \mathcal{A}_i$ for each $i \in \{0, \dots, d\}$ is called a **strong** $\mathcal{A}$**-filtration** of C.

Remark 5.38. Let $\mathcal{A} = (A_0, \dots, A_d)$ be a variable assignment stack and let $C \subseteq \mathbb{L}_n$ be a reason clause under A_d.

(a) Every *strong* $\mathcal{A}$-filtration $\mathcal{B} = (B_0, \dots, B_d, B_\infty)$ satisfies

$$\begin{array}{ccccccc}
V_C \cap \mathcal{A}_0 & \subseteq & \cdots & \subseteq & V_C \cap \mathcal{A}_{d-1} & \subseteq & V_C \cap \mathcal{A}_d \\
\| & & & & \| & & \| \\
\langle B_0 \rangle_{\mathbb{F}_2} & \subseteq & \cdots & \subseteq & \langle B_0 \cup \dots \cup B_{d-1} \rangle_{\mathbb{F}_2} & \subseteq & \langle B_0 \cup \dots \cup B_d \rangle_{\mathbb{F}_2}
\end{array}$$

(b) Every $\mathcal{A}$-filtration $\mathcal{B} = (B_0, \dots, B_d, B_\infty)$ satisfies

$$\begin{array}{ccccccc}
V_C \cap \mathcal{A}_0 & \subseteq & \cdots & \subseteq & V_C \cap \mathcal{A}_{d-1} & \subseteq & V_C \cap \mathcal{A}_d \\
\cup| & & & & \cup| & & \| \\
\langle B_0 \rangle_{\mathbb{F}_2} & \subseteq & \cdots & \subseteq & \langle B_0 \cup \dots \cup B_{d-1} \rangle_{\mathbb{F}_2} & \subseteq & \langle B_0 \cup \dots \cup B_d \rangle_{\mathbb{F}_2}
\end{array}$$

5.4.1 Filtrations and Watching Structures. Before we discuss methods to *combine* filtrations efficiently, let us show how watching structures and filtrations can be converted to one another.

Remark 5.39 (Filtrations from Watching Structures)**.** Let $\mathcal{A} = (A_0, \dots, A_d)$ be a variable assignment stack, and let $C \subseteq \mathbb{L}_n$ be a non-trivial reason clause under A_d. Furthermore, let $W = \big((\ell_1, \dots, \ell_k),\ (i_1, j_1),\ (i_2, j_2)\big) \xrightarrow{\mathcal{A}} C$ be an $\mathcal{A}$-watching structure for C. Then Proposition 5.28.b and (ws_5) yield $\ell_j|_{A_d} = 0$. For $i \in \{0, \dots, d\}$ let

$$B_i \;=\; \big\{\ell_j \mid j \in \{1, \dots, k\} \setminus \{j_1\},\ \delta_{\mathcal{A}}(\ell_j) = i\big\}.$$

If $\ell_{j_1}|_{A_d} = 0$, then also add ℓ_{j_1} to $B_{\delta_{\mathcal{A}}(\ell_{j_1})}$ and let $B_\infty = \emptyset$; otherwise let $B_\infty = \{\ell_{j_1}\}$. Then, due to ($\mathrm{ws}_1$) and Proposition 5.28.b, the tuple $\mathcal{B} = (B_0, \dots, B_d, B_\infty)$ is an $\mathcal{A}$-filtration of C. Moreover, if $\mathcal{A}$ is a stepwise variable assignment stack, then $\mathcal{B}$ is a watchable $\mathcal{A}$-filtration.

If an $\mathcal{A}$-filtration $\mathcal{B}$ arises from an $\mathcal{A}$-watching structure W using this construction, we also write $\mathcal{B} = \mathsf{WatchToFiltr}_{\mathcal{A}}(W)$.

The above construction can be implemented efficiently if the *dl-count tuples* as in Remark 5.33 are stored for each lineral of C.

Example 5.40. Consider the $\mathcal{A}$-watching structure

$$W \;=\; \Big(\big(\,x_1 + x_2 + \underline{x_4},\ x_1,\ x_2 + \underline{x_3} + 1,\ x_4 + 1\,\big),\ (3,3),\ (4,1)\Big)$$

from Example 5.36. Because of $\delta_{\mathcal{A}}(x_1 + x_2 + x_4) = \delta_{\mathcal{A}}(x_4 + 1) = 3$ and $\delta_{\mathcal{A}}(x_1) = 1$, $\mathsf{WatchToFiltr}_{\mathcal{A}}(W)$ computes the $\mathcal{A}$-filtration $\mathcal{B} = \big(B_0, B_1, B_2, B_3, B_\infty\big)$ with

$$B_0 = \emptyset,\ \ B_1 = \{x_1\},\ \ B_2 = \emptyset,\ \ B_3 = \{x_1 + x_2 + x_4,\ x_4 + 1\},\ \text{ and } B_\infty = \{x_2 + x_3 + 1\}.$$

Moreover, $\mathcal{B}$ is a watchable $\mathcal{A}$-filtration, as $\lambda(\mathcal{B}) = \infty$ and $\#B_\infty = 1$.

The next remark shows how reason clause decompositions can be obtained from filtrations.

Remark 5.41. Let $\mathcal{A} = (A_0, \dots, A_d)$ be a variable assignment stack, let $C \subseteq \mathbb{L}_n$ be a reason clause under A_d, and let $\mathcal{B} = (B_0, \dots, B_d, B_\infty)$ be an $\mathcal{A}$-filtration of C.

(a) The clause C is equivalent to the clause $1 + (B_0 \cup \dots \cup B_d \cup B_\infty)$. In particular, a reason clause decomposition of V_C under A_d is given by

$$V_C \;=\; \langle B_0 \cup \dots \cup B_d \rangle_{\mathbb{F}_2} \oplus \langle B_\infty \rangle_{\mathbb{F}_2}.$$

This also shows that C is a conflict clause under A_d if and only if $B_\infty = \emptyset$.

(b) If $B_\infty \neq \emptyset$, i.e., C is not a conflict clause under A_d, then $\lambda(\mathcal{B}) = \infty$ and $\#B_\infty = 1$ shows that $\mathcal{B}$ is a watchable $\mathcal{A}$-filtration.

(c) If $B_\infty = \emptyset$ and $\mathcal{B}$ is a watchable $\mathcal{A}$-filtration with $\lambda(\mathcal{B}) \neq \infty$, then there is some $\ell \in \mathbb{L}_n$ with $B_{\lambda(\mathcal{B})} = \{\ell + 1\}$. Moreover, C is a reason clause under $A_{\lambda(\mathcal{B})-1}$ with reason lineral ℓ. A reason clause decomposition of V_C is therefore given by

$$V_C \;=\; \langle B_0 \cup \dots \cup B_{\lambda(\mathcal{B})-1} \rangle_{\mathbb{F}_2} \oplus \langle \ell + 1 \rangle_{\mathbb{F}_2}.$$

As our terminology suggests, a watching structure can be obtained from a *watchable* filtration. Although this may seem a limiting factor, Remark 5.41.b indicates that many filtrations are trivially watchable. Moreover, there is an algorithm which can *refine* any given $\mathcal{A}$-filtration under a *stepwise variable assignment stack* $\mathcal{A}$. It can be tuned to produce also watchable and strong $\mathcal{A}$-filtrations. In the latter case, the algorithm essentially performs a *Gaußian Elimination*, where the variables with the highest decision level are chosen as pivots.

Algorithm 9: ReFiltr – Refine Filtrations

Input : A stepwise variable assignment stack $\mathcal{A} = (A_0, \dots, A_d)$, an $\mathcal{A}$-filtration $\mathcal{B}$ for a reason clause $C \subseteq \mathbb{L}_n$ under A_d, $c \in \{1, \dots, d\}$.

Output : An $\mathcal{A}$-filtration $\mathcal{B}' = (B'_0, \dots, B'_d, B'_\infty)$ of C with $\#B'_i \leq 1$ for all $i \in \{c, \dots, d\}$.

1 Let $(B'_0, \dots, B'_d, B'_\infty) = \mathcal{B}$.
2 **for** $i = d$ **to** c **do**
3 **if** $\#B'_i > 1$ **then**
4 Let $\ell' \in B'_i$.
5 **for** $\ell \in B'_i \setminus \{\ell'\}$ **do**
6 **if** $\ell + \ell' \neq 0$ **then** adjoin $\ell + \ell'$ to $B'_{\delta_{\mathcal{A}}(\ell+\ell')}$.
7 Let $B'_i = \{\ell'\}$.
8 **return** $(B'_0, \dots, B'_d, B'_\infty)$.

Proposition 5.42. Let $\mathcal{A} = (A_0, \dots, A_d)$ be a stepwise variable assignment stack, let $\mathcal{B}$ be an $\mathcal{A}$-filtration of a reason clause $C \subseteq \mathbb{L}_n$ under A_d, and let $c \in \{1, \dots, d\}$. Then ReFiltr is an algorithm. It returns $\mathcal{B}' = (B'_0, \dots, B'_d, B'_\infty) = \mathsf{ReFiltr}_{\mathcal{A}}(\mathcal{B}, c)$ with the following properties.

(a) $\mathcal{B}'$ is an $\mathcal{A}$-filtration of C with $\#B'_i \leq 1$ for $i \in \{c, \dots, d\}$.

(b) If $c \leq \lambda(\mathcal{B})$, then $\mathcal{B}'$ is a watchable $\mathcal{A}$-filtration of C.

(c) If $c = 1$, then $\mathcal{B}'$ is a strong $\mathcal{A}$-filtration of C.

Proof. Let us show the correctness of the procedure. Note that the conditions (3) and (4) from Definition 5.37 are equivalent to $B'_i = \{\ell \in B'_0 \cup \dots \cup B'_d \mid \delta_{\mathcal{A}}(\ell) = i\}$ for all $i \in \{1, \dots, d\}$. The latter is clearly satisfied whenever $\ell + \ell'$ is adjoined to $B'_{\delta_{\mathcal{A}}(\ell+\ell')}$ in line 6. Additionally, the set $B'_0 \cup \dots \cup B'_d$ still generates $V_C \cap \mathcal{A}_d$ after line 7, i.e., $\mathcal{B}$ is an $\mathcal{A}$-filtration of C. Moreover, because $\mathcal{A}$ is a stepwise assignment stack, there is a unique variable x with $\delta_{\mathcal{A}}(x) = i$. This implies $x \in \bigcap_{\ell \in B'_i} \operatorname{Vars}(\ell)$ and we get $x \notin \operatorname{Vars}(\ell + \ell')$ in line 6. Thus, we have $\delta_{\mathcal{A}}(\ell + \ell') < i$, i.e., the set B'_i is not enlarged at any later point during execution. Therefore, $\#B'_i \leq 1$ for all $i \in \{c, \dots, d\}$ in line 8. This shows parts (a) and (b), and the termination of the procedure.

For (c), let $\#B'_j \leq 1$ for all $j \in \{1, \dots, d\}$. As $\mathcal{B}'$ is an $\mathcal{A}$-filtration of C we have $B'_i \subseteq \mathcal{A}_i$ and $B'_i \subseteq V_C$. Thus, it suffices to show $(V_C \cap \mathcal{A}_i) \setminus \mathcal{A}_{i-1} \subseteq \langle B'_0 \cup \dots \cup B'_i \rangle_{\mathbb{F}_2}$ for all $i \in \{0, \dots, d\}$. Therefore, let $\ell \in (V_C \cap \mathcal{A}_i) \setminus \mathcal{A}_{i-1}$. Clearly this yields $\delta_{\mathcal{A}}(\ell) = i$. Because of $V_C \cap \mathcal{A}_i \subseteq V_C \cap \mathcal{A}_d = \langle B'_0 \cup \dots \cup B'_d \rangle_{\mathbb{F}_2}$, there is $j \in \{i, \dots, d\}$ minimal with $\ell \in \langle B'_0 \cup \dots \cup B'_j \rangle_{\mathbb{F}_2}$. With $\#B'_j \leq 1$ we get $B'_j = \{\ell_j\}$ for some $\ell_j \in \mathbb{L}_n$ with $\ell \in \ell_j + \langle B'_0 \cup \dots \cup B'_{j-1} \rangle_{\mathbb{F}_2}$. Now, $\ell_j \in B'_j$ implies $\delta_{\mathcal{A}}(\ell_j) = j$. Furthermore, as $\mathcal{A}$ is a variable assignment stack, there is also an indeterminate $x \in \operatorname{Supp}(\ell_j)$ with $\delta_{\mathcal{A}}(x) = j$. Now notice that every lineral $\ell' \in \langle B'_0 \cup \dots \cup B'_{j-1} \rangle_{\mathbb{F}_2} \subseteq \mathcal{A}_{j-1}$ satisfies $\delta_{\mathcal{A}}(\ell') < j$. In particular, this yields $x \notin \operatorname{Supp}(\ell')$. Finally, $\ell \in \ell_j + \langle B'_0 \cup \dots \cup B'_{j-1} \rangle_{\mathbb{F}_2}$ implies $x \in \operatorname{Supp}(\ell)$ and we get $i = \delta_{\mathcal{A}}(\ell) \geq \delta_{\mathcal{A}}(x) = j \geq i$, i.e., $i = j$ and we have $\ell \in \langle B'_0 \cup \dots \cup B'_i \rangle_{\mathbb{F}_2}$. □

Because of Remark 5.41.b and Proposition 5.42.b, a given $\mathcal{A}$-filtration $\mathcal{B}$ can be *refined* to a watchable $\mathcal{A}$-filtration. In particular, this filtration is given by

$$\mathsf{WatchableFiltr}_{\mathcal{A}}(\mathcal{B}) = \begin{cases} \mathsf{ReFiltr}_{\mathcal{A}}(\mathcal{B}, \lambda(\mathcal{B})) & \text{if } B_\infty = \emptyset, \\ \mathcal{B} & \text{otherwise.} \end{cases}$$

Finally, let us show how watchable filtrations give rise to watching structures. In order to simplify the description, from now on, we set $\delta_{\mathcal{A}}(\ell) = \infty$ for all $\ell \notin \mathcal{A}_d \cup (1 + \mathcal{A}_d)$.

Proposition 5.43 (Watching Structures from Watchable Filtrations)**.** Let $\mathcal{A} = (A_0, \dots, A_d)$ be a variable assignment stack, let $C \subseteq \mathbb{L}_n$ be a non-trivial reason clause under A_d with $\#C > 1$ and $C \not\equiv \bot$, and let $\mathcal{B} = (B_0, \dots, B_d, B_\infty)$ be a watchable $\mathcal{A}$-filtration of C. Consider the following sequence of instructions.

(1) Write $\{\ell_1, \dots, \ell_k\} = B_0 \cup \dots \cup B_d \cup B_\infty$.

(2) Choose $(i_1, j_1) \in \{1, \dots, n\} \times \{1, \dots, k\}$ with $\ell_{j_1} \in B_{\lambda(\mathcal{B})}$, $x_{i_1} \in \mathrm{Vars}(\ell_{j_1})$, and $\delta_{\mathcal{A}}(x_{i_1}) = \delta_{\mathcal{A}}(\ell_{j_1})$.

(3) Let $k \in \{0, \dots, d\}$ be maximal with $B_k \neq \emptyset$ and $k < \lambda(\mathcal{B})$.

(4) Choose $(i_2, j_2) \in \{1, \dots, n\} \times \{1, \dots, k\}$ with $\ell_{j_2} \in B_k$, $x_{i_1} \in \mathrm{Vars}(\ell_{j_2})$, and $\delta_{\mathcal{A}}(x_{i_2}) = \delta_{\mathcal{A}}(\ell_{j_2})$. If $x_{i_2} \in \mathrm{Vars}(\ell_{j_1})$, then replace ℓ_{j_1} by $\ell_{j_1} + \ell_{j_2}$.

(5) Return the tuple $W = \big((\ell_1, \dots, \ell_k),\ (i_1, j_1),\ (i_2, j_2)\big)$.

This is an algorithm, denoted by WatchableFiltrToWatch, which computes an $\mathcal{A}$-watching structure for C.

Proof. By definition, we have $V_C = \langle B_0 \cup \dots \cup B_\infty \rangle_{\mathbb{F}_2} = \langle \ell_1, \dots, \ell_k \rangle_{\mathbb{F}_2}$. Thus, W satisfies (ws_1). After step (4) we also have (ws_2). The construction furthermore ensures (ws_3) and (ws_4) for all A_i, because of $\delta_{\mathcal{A}}(x_{i_1}) = \delta_{\mathcal{A}}(\ell_{j_1}) = \lambda(\mathcal{B}) > k = \delta_{\mathcal{A}}(x_{i_2}) = \delta_{\mathcal{A}}(\ell_{j_2})$. Property ($\mathrm{ws}_5$) follows from the fact that k is the second-highest index $i \in \{0, \dots, d, \infty\}$ with $B_i \neq \emptyset$, i.e., ℓ_{j_2} has the second highest decision level after ℓ_{j_1}, and all ℓ_j satisfy $\ell_j|_{A_d} = 0$. Altogether, this shows that W is an $\mathcal{A}$-watching structure for C. □

5.4.2 Clause Learning using Filtrations of Reason Clauses. Now, we can shift our focus towards using filtrations in an implementation of LinTrailRes. For this, we need to be able to *extend* given filtrations appropriately. In particular, we mimic Proposition 4.32 using filtrations. Recall that it shows how two reason clauses can be combined to give a reason clause for the sum of their implied linerals.

Proposition 5.44. Let $\mathcal{A} = (A_0, \dots, A_d)$ be a variable assignment stack and let $u_1, u_2 \in \mathbb{L}_n \setminus \{0\}$ be distinct linerals. Let $k \in \{0, \dots, d\}$. For $i \in \{1, 2\}$ let $C_i \subseteq \mathbb{L}_n$ be a reason clause for u_i under A_k and let $\mathcal{B}^{(i)} = (B_0^{(i)}, \dots, B_d^{(i)}, B_\infty^{(i)})$ be a watchable $\mathcal{A}$-filtration of C_i with $B^{(i)}_{\lambda(\mathcal{B}^{(i)})} = \{\ell_i + 1\}$.

(a) For $i \in \{1, 2\}$ let $\tilde{\mathcal{B}}^{(i)} = (\tilde{B}_0^{(i)}, \dots, \tilde{B}_d^{(i)}, \tilde{B}_\infty^{(i)})$ with

$$\tilde{B}_j^{(i)} \;=\; \begin{cases} \emptyset & \text{if } j = \lambda(\mathcal{B}^{(i)}), \\ B_j^{(i)} & \text{otherwise.} \end{cases}$$

Then $V_{C_i} \cap \mathcal{A}_k = \langle \tilde{B}_0^{(i)} \cup \dots \cup \tilde{B}_d^{(i)} \cup \tilde{B}_\infty^{(i)} \rangle_{\mathbb{F}_2}$.

(b) Let $\tilde{\mathcal{B}} = (\tilde{B}_0, \dots, \tilde{B}_d, \tilde{B}_\infty)$ with

$$\tilde{B}_j \;=\; \begin{cases} \tilde{B}_j^{(1)} \cup \tilde{B}_j^{(2)} \cup \{\ell_1 + \ell_2 + 1\} & \text{if } j = \delta_{\mathcal{A}}(\ell_1 + \ell_2 + 1), \\ \tilde{B}_j^{(1)} \cup \tilde{B}_j^{(2)} & \text{otherwise.} \end{cases}$$

Then $R = 1 + \big(\tilde{B}_0 \cup \dots \cup \tilde{B}_d \cup \tilde{B}_\infty\big)$ is a reason clause for $u_1 + u_2$ under A_k with

$$V_R \;=\; \big(V_{C_1} \cap \mathcal{A}_k\big) \;+\; \big(V_{C_2} \cap \mathcal{A}_k\big) \;+\; \langle \ell_1 + \ell_2 + 1 \rangle_{\mathbb{F}_2},$$

and $\tilde{\mathcal{B}}$ is an $\mathcal{A}$-filtration of R.

Proof. Part (a) follows from Remark 5.41 and the fact that $\mathcal{B}$ is a watchable $\mathcal{A}$-filtration. For (b) it suffices to note that by (a) and the definition of R and $\tilde{\mathcal{B}}$ we have $V_R = \big(V_{C_1} \cap \mathcal{A}_k\big) + \big(V_{C_2} \cap \mathcal{A}_k\big) + \langle \ell_1 + \ell_2 + 1 \rangle_{\mathbb{F}_2}$ and $\tilde{\mathcal{B}}$ is an $\mathcal{A}$-filtration of R. Finally, Proposition 4.8 shows that R is indeed a reason clause for $(\ell_1 + \ell_2)|_{A_k} = u_1 + u_2$ under A_k. □

Remark 5.45. In Remark 5.7, we presented two modifications of GCP$^{\text{lite}}$ which improve the deductive power. By Remark 5.9 executing such modified variants yield a lite lineral trail by adding reason clauses of the sum of implied linerals to our XNF formula F. Because of WatchToFiltr, WatchableFiltr, and WatchableFiltrToWatch, Proposition 5.44.b allows us to efficiently compute watching structures for these clauses given watching structures for all involved clauses.

In the context of LinTrailRes, we use Proposition 5.44 in the following setting.

Remark 5.46. Let $\mathcal{A} = (A_0, \dots, A_d)$ be a stepwise assignment stack and $\ell \in \mathbb{L}_n$ with $A_k = A_{k-1} \cup \{\ell\}$. Let $C \subseteq \mathbb{L}_n$ be a conflict clause under A_k, but not under A_{k-1}; let $R \subseteq \mathbb{L}_n$ be a reason clause for ℓ under A_{k-1}. Furthermore, let $\mathcal{B} = (B_0, \dots, B_d, B_\infty)$ and $\mathcal{B}' = (B'_0, \dots, B'_d, B'_\infty)$ be watchable $\mathcal{A}$-filtrations of C and R, respectively.

(a) Since C is a conflict clause under A_k, we have $B_{k+1} = \dots = B_d = B_\infty = \emptyset$. By Proposition 4.37.a, C is a reason clause for $\ell + 1$ under A_{k-1}, i.e., we have $B_k \neq \emptyset$ and $\lambda(\mathcal{B}) = k$. Furthermore, R is a reason clause for ℓ under A_{k-1} and $\ell \in A_k$. Therefore, we get $B'_k = \dots = B'_d = \emptyset$ and $B'_\infty \neq \emptyset$.

(b) Write $B_k = \{\ell_1\}$ and $B'_\infty = \{\ell_2\}$. An $\mathcal{A}$-filtration $\tilde{\mathcal{B}} = (\tilde{B}_0, \dots, \tilde{B}_d, \tilde{B}_\infty)$ of a conflict clause C' under A_{k-1}, i.e., a reason clause for $1 = \ell + (\ell + 1)$, can be constructed with Proposition 5.44. In particular, we have

$$\tilde{B}_j = \begin{cases} \emptyset & \text{if } j \geq k, \\ B_j \cup B'_j \cup \{\ell_1 + \ell_2 + 1\} & \text{if } j = \delta_{\mathcal{A}}(\ell_1 + \ell_2 + 1), \\ B_j \cup B'_j & \text{otherwise.} \end{cases}$$

Then $\tilde{\mathcal{B}}$ is an $\mathcal{A}$-filtration of a conflict clause C' under A_{k-1}.

Now we are ready to present LinTrailRes*, a version of LinTrailRes, in Algorithm 10 which works directly with filtrations and watching structures. It outputs the same asserting clause but as a watching structure which can be integrated directly into the clause database of an implementation of CDXCL$^{\text{lite}}$.

Proposition 5.47. Let $\mathcal{L}$ be a lite conflict XNF solver state for an XNF formula F with associated variable assignment stack $\mathcal{A}$ and associated stepwise variable assignment stack $\mathcal{A}'$. Then LinTrailRes* is an algorithm which returns an $\mathcal{A}'$-watching structure for an $\mathcal{A}$-asserting clause $C \subseteq \mathbb{L}_n$ with a polynomial-size proof $F \vdash_{\mathsf{XLIN}} C$.

Proof. Write $\mathcal{A}' = (A_0, \dots, A_s)$, and denote the last $k + 1$ entries by $A'_0, \dots, A'_{k+1}$. Then $A'_i = A_{i-1} \cup \{\ell_i\}$, for all $i \in \{1, \dots, k\}$. Let us show that LinTrailRes* is equivalent to LinTrailRes, only that it returns an $\mathcal{A}'$-watching structure. To see this, identify $1 + B$ with G, V with $\langle G \rangle_{\mathbb{F}_2}$, and L' with A_{i-1}.

The variable x_{i_1} in line 5 has the highest decision level with respect to $\mathcal{A}'$ in the first watched lineral of W. Therefore, it has the highest decision level among all variables of the clause C. As a consequence, C' is a conflict clause under A'_i if and only if x_{i_1} is assigned by A'_i. In other words, the condition in line 5 is true if only if C' is *not* a conflict clause under A'_{i-1}. This corresponds directly to the condition in line 5 of LinTrailRes.

As $\mathcal{A}'$ is a stepwise assignment stack and C_i is a reason clause under A_{i-1}, the output of WatchToFiltr in line 6 has the claimed form due to Remark 5.46.a. Part (c) of the same remark shows that lines 6-10 compute $\mathcal{B}$ as a watchable $\mathcal{A}'$-filtration of the conflict clause C' under A_{i-1} as in Proposition 4.37. This mimics the computation of lines 6-9 in LinTrailRes. The condition $\delta_{\mathcal{A}}(\ell_{j_2}) < d$ in line 12 is equivalent to C' being $\mathcal{A}$-asserting due to Proposition 5.30, i.e., it corresponds to line 10 in LinTrailRes.

Finally, the claims follow from the proof of LinTrailRes (Proposition 4.38). □

To get an efficient implementation of LinTrailRes*, consider the following remark.

Remark 5.48 (Implementation Details)**.**

Algorithm 10: LinTrailRes* – Clause Learning by Resolving along Lineral Trail

Input : A lite conflict XNF solver state $\mathcal{L}$ with associated variable assignment stack $\mathcal{A} = (A_0, \ldots, A_d)$, the associated stepwise variable assignment stack $\mathcal{A}'$, and $\mathcal{A}'$-watching structures for all clauses $C \in F$.

Output : An $\mathcal{A}'$-watching structure W for an $\mathcal{A}$-asserting XNF clause $C \subseteq \mathbb{L}_n$ with $F \vdash_{\mathsf{XLIN}} C$.

1 Let $[\ell_1^{C_1}, \ldots, \ell_k^{C_k}, 1^C]$ be the d-th *literal* trail of $\mathcal{L}$.
2 Let $W = \big(G,\ (i_1, j_1),\ (i_2, j_2)\big) \xrightarrow{\mathcal{A}'} C$, and let $W_i \xrightarrow{\mathcal{A}'} C_i$ for $i \in \{1, \ldots, k\}$.
3 Let $\mathcal{B} = (B_0, \ldots, B_s, B_\infty) = \mathsf{WatchToFiltr}_{\mathcal{A}'}(W)$.
4 **for** $i = k$ **to** 1 **do**
5 **if** $\mathrm{Vars}(\ell_i) = \{x_{i_1}\}$ **then**
6 Let $(B'_0, \ldots, B'_s, \{\ell'_i + 1\}) = \mathsf{WatchToFiltr}_{\mathcal{A}'}(W_i)$.
7 **for** $j = 0$ **to** $\lambda(\mathcal{B}) - 1$ **do** adjoin B'_j to B_j.
8 Write $B_{\lambda(\mathcal{B})} = \{\ell + 1\}$, and let $B_{\lambda(\mathcal{B})} = \emptyset$.
9 Adjoin $\ell'_i + \ell + 1$ to $B_{\delta_{\mathcal{A}'}(\ell'_i + \ell + 1)}$.
10 Compute $\mathcal{B} = (B_0, \ldots, B_s, B_\infty) = \mathsf{WatchableFiltr}_{\mathcal{A}'}(\mathcal{B})$.
11 Compute $W = (G,\ (i_1, j_1),\ (i_2, j_2)) = \mathsf{WatchableFiltrToWatch}_{\mathcal{A}'}(\mathcal{B})$.
12 **if** $\delta_{\mathcal{A}}(\ell_{j_2}) < d$ **then return** W.
13 **return** W.

(a) In lines 5-11 we only use the first watched variable from the $\mathcal{A}'$-watching structure $W = \big(G,\ (i_1, j_1),\ (i_2, j_2)\big)$. Thus, it suffices to compute only this part of W in line 11. A close look at WatchableFiltrToWatch also reveals that, in fact, it can be derived from $B_{\lambda(\mathcal{B})}$ quickly. Only when the algorithm returns in line 12 or 13 we need to compute the other components of the watching structure W.

(b) The filtration $\mathcal{B}$ does not need to be stored explicitly while it changes during the iterations of the loop (lines 5-12). Instead, it suffices to extend the watching structure $W = \big((\ell_1, \ldots, \ell_k),\ (i_1, j_1),\ (i_2, j_2)\big)$ in line 3 with a map $\varphi\colon \{0, \ldots, d\} \longrightarrow \mathcal{P}(\mathbb{N})$, such that $\mathcal{B} = (\{\ell_i \mid i \in \varphi(0)\}, \ldots, \{\ell_i \mid i \in \varphi(d)\},\ \emptyset)$. Then lines 7 and 9 correspond to the addition of new linerals to the tuple of generators of W as well as adjoining the corresponding indices to appropriate images of φ; and in line 8 we set $\varphi\big(\lambda(\mathcal{B})\big) = \emptyset$. The application of WatchableFiltr in line 10 amounts to rewriting $\#\varphi\big(\lambda(\mathcal{B})\big) - 1$ linerals and distributing all but one index from $\varphi\big(\lambda(\mathcal{B})\big)$ to other images of φ.

Our suggestion is to represent the map φ internally as a sorted list of (non-empty) sets. In this way, the leading level $\lambda(\mathcal{B})$ along with $\varphi\big(\lambda(\mathcal{B})\big)$ can be found in constant time.

(c) To enable fast construction of the filtrations in lines 3 and 6, we store the decision level $\delta_{\mathcal{A}'}(\ell)$ with respect to the *stepwise* variable assignment stack $\mathcal{A}'$ for each lineral of a watching structure W. It is cheap to compute this value during UpdWatchStruct (lines 7 and 11) and allows the filtrations in LinTrailRes* to be constructed efficiently.

In view of our *dl-counting scheme*, see Remark 5.33, an implementation should store information with respect to two different levels: the level in which ℓ is known to evaluate to zero with respect to the associated variable assignment stack $\mathcal{A}$ *and* with respect to the associated *stepwise* variable assignment stack $\mathcal{A}'$.

(d) The longer the literal trail in line 1 is, the higher the number of different linerals in the watching structure W can become. Contrary to the fact that the degree of the corresponding clause might still be comparatively

small. This can happen, as the linerals in G need not be linearly independent. To cope with redundant linerals and avoid too much memory consumption, we suggest the following heuristic:

Whenever the number of linerals in G exceeds 256 in line 11, use a modified version of $\mathsf{ReFiltr}_{\mathcal{A}'}(\mathcal{B}, 1)$ to ensure that $\#B_i \leq 4$ for all $i \in \{0, \ldots, s\}$ (by changing the condition in line 3 of ReFiltr). Similarly, one should reduce the representation before returning the watching structure in line 13.

This closes our discussion on data structures for an efficient implementation of $\mathsf{CDXCL}^{\mathsf{lite}}$.

6 Experiments with XOR-Rich Problems

The XNF-based SAT solving algorithm $\mathsf{CDXCL}^{\mathsf{lite}}_{\mathsf{GJE}}$ has been implemented in C++ using the data structures from the previous section as the solver Xorricane. It features a few established decision, restart, and clause deletion heuristics known from CNF-based solvers, as well as the inprocessing suggestion of Remark 5.24 and the preprocessing methods of the graph-based 2-XNF solver 2-Xornado (Andraschko et al. 2024).

A detailed description of the methods which were employed and their parameters can be found in Appendix A. Here we only want to comment on some areas where the implementation of Xorricane can still be improved.

Remark 6.1. Although Xorricane extends $\mathsf{CDXCL}^{\mathsf{lite}}$ with some established heuristics and methods known from modern CNF-based SAT solvers, it has many shortcomings in direct comparison with state-of-the-art CDCL implementations. To list just a few, its heuristics are not properly fine-tuned, it offers little in- and preprocessing techniques, there is no clause minimization, no chronological backtracking, no switching between *stable* and *focused* mode, no local search, and the implementation does not consider cache optimization.

Given (Biere, Fazekas, et al. 2020; Biere and Fröhlich 2018; Chu et al. 2010; Eén and Sörensson 2004; Hölldobler et al. 2010; Järvisalo, Heule, et al. 2012; Lewis et al. 2005; Soos 2023), this list provides many areas each with the potential for a significant performance improvement.

Apart from Xorricane and 2-Xornado, no other XNF-based solvers have been developed so far. Thus, they can only be compared with established algebraic and logic solvers. In particular, we consider solvers for problems in ANF, XNF, 2-XNF, CNF-XOR, and CNF. This requires to convert the XNF benchmark instances to all these different input types.

Remark 6.2 (Conversions between Normal Forms)**.**

(a) Given a set of Boolean polynomials $S \subseteq \mathbb{B}_n$ in ANF, we can obtain a 2-XNF encoding F of S using (Andraschko et al. 2024, Algorithm 2) or (Andraschko et al. 2024, Algorithm 3). In particular, then F is also an XNF encoding of S.

(b) Conversely, if an XNF formula F is given, it can be converted to ANF by multiplying the linerals of each clause (see Remark 2.4). Moreover, a 2-XNF encoding can be computed with (Andraschko et al. 2024, Algorithm 1).

(c) Starting from an XNF encoding F, we get a CNF-XOR encoding G using the construction of Definition B.1.a, which adds linearly many new variables and clauses.

To get a CNF encoding, it now suffices to use the CNF clauses of G and add CNF encodings of its XOR clauses. An exponential increase in the number of clauses during the latter encodings can be avoided by cutting each XOR after a chosen number of literals. To be precise, we use *pooled* encodings of XOR clauses, as described in (Nawrocki et al. 2021), and use five as our *cutting number*. In view of (Bard et al. 2007; Bulygin and Buchmann 2011; Nawrocki et al. 2021; Soos and Meel 2019), this choice seems adequate. Ultimately, a CNF encoding whose size is linear in the size of F can be constructed.

Altogether, given XNF or ANF instances, the above allows us to construct encodings in all other normal forms.

6.1 Competing Solving Algorithms

Before introducing our benchmark suites, let us briefly present the CNF-, CNF-XOR-, XNF-, and ANF-based solvers used for our experiments.

CNF. Over the last three decades, the continuous improvement of CDCL-based SAT solvers has borne fruit, see (Biere, Fleury, et al. 2023). In particular, the performance of today's SAT solvers far exceeds that of the solvers of previous decades. This is mainly due to the fine-tuning of the heuristics and the addition of various pre- and inprocessing techniques. To offer a fair comparison, we do not only compare against state-of-the-art solvers Kissat (version *sc2024*) (Biere, Fazekas, et al. 2020), winner of the SAT Competition 2024, but also chose to include *older* and less powerful solvers that lack the latest modern techniques. To this end, we consider Kissat `–plain`, a version of Kissat where advanced solving techniques are deactivated, as well as Glucose (version *2.0*) (Audemard and Simon 2009), winner of the SAT Competition 2011 in the *application track*. Just like Xorricane, these solvers use variants of the VSIDS decision and (dynamic) restart heuristics, but also additionally employ in- and preprocessing methods like *clause minimization* (Beame, Kautz, et al. 2004; Eén and Sörensson 2004; Sörensson and Biere 2009), *on-the-fly subsumption* (Hamadi et al. 2009; H. Han and Somenzi 2009; Soos, Nohl, et al. 2009), or *vivification* (Li et al. 2020), and have mature implementations with special attention to cache-optimality. Nonetheless, from an algorithmic perspective these solvers are more comparable to Xorricane.

Moreover, we include Bosphorus (version *3.0.0*) (Choo et al. 2019), which combines algebraic and logical reasoning by switching between both worlds repeatedly. On the algebraic side it uses XL (Courtois and Patarin 2003) and ElimLin (Courtois, Sepehrdad, et al. 2012), whereas the logical side is powered by CryptoMiniSat (Soos, Nohl, et al. 2009), a CNF-XOR SAT solver. It can read both ANF and CNF inputs. We denote the variant that reads CNF inputs by Bosphorus-C.

CNF-XOR. In (C.-S. Han and Jiang 2012; Soos 2010; Soos, Gocht, et al. 2020; Soos and Meel 2019; Soos, Nohl, et al. 2009) a generalization of the CDCL framework for CNF-XOR formulae is presented, which we denote by $\mathsf{CDCL}^{\mathsf{XOR}}_{\mathsf{GJE}}$. It uses $\mathsf{BCP}^{\mathsf{XOR}}_{\mathsf{GJE}}$ for propagation, and weakens XOR clauses to corresponding CNF clauses whenever they are used during conflict analysis. The theoretical analysis is supported by the practical solver CryptoMiniSat (Soos, Nohl, et al. 2009) which offers more than 20 in- and preprocessing techniques and a highly optimized implementation. In our experiments we use version *5.11.23*. Similar to the above, we also compare with an older version of the solver, namely CryptoMiniSat 2.4, winner of SAT RACE 2010. Since the default configuration crashed on almost half of the benchmark instances, we use the optional `–noxorsubs` argument, which disables *subsumption* of XOR clauses and no longer crashes prematurely.

To our knowledge, xnfSAT (Nawrocki et al. 2021) is the only other CNF-XOR SAT solver. Despite its name, it can only parse CNF-XOR formulae and is based on a local search algorithm. Due to its poor performance on small instances, see (Andraschko et al. 2024), it is excluded from our benchmarks.

XNF. Apart from 2-Xornado (Andraschko et al. 2024) and Xorricane there exist no other XNF-based solvers so far. We consider 2-Xornado (version *1.0.3*) with the MaxReach decision heuristic and two variants of Xorricane. Both use the dynamic scheduling of our inprocessing method (`–impl-lits -1`) but differ in the choice of the deletion heuristic, `avg_util` and `lbd` (for details, see Appendix A). The solvers are abbreviated as Xorricane `avg_util` and Xorricane `lbd`.

By Remark 5.6, it is possible to *simulate* GCP within $\mathsf{BCP}^{\mathsf{XOR}}_{\mathsf{GJE}}$ for variable assignments by using an appropriate CNF-XOR encoding of XNF input problems. To this end, we also consider CryptoMiniSat `–ext`, a variant of the solver CryptoMiniSat, which takes a 2-XNF formula F as input and applies it to the extended trivial CNF-XOR encoding of F, see Definition B.1.b.

ANF. An efficient implementation for the computation a Boolean Gröbner basis is provided by PolyBoRi (Brickenstein and Dreyer 2009). It allows the construction of a solver, which takes polynomials in ANF as input and returns a common zero, i.e., a satisfying assignment of the corresponding logical representation. Other algebraic methods like XL and ElimLin are not complete in general and lack an accessible implementation. Hence they are not included separately in our comparison. However, we include the solver Bosphorus which also uses these techniques. We denote the variant which reads ANF inputs by Bosphorus-A.

In order to put the above-mentioned algorithms to the test, we consider a random and a cryptographic benchmark suite. Each suite consists of 150 instances related to three types of problems, each with varying hardness. The following two subsections describe these benchmark suites in detail and present experimental results. The performance of the solvers is measured by the number of instances that can be solved within a timeout of *1000 seconds*. However, we also include results where the logical solvers are additionally limited to 10^6 *decisions*. We use cactus plots to visualize the distribution of solving times and decisions, as defined in (Brain, Davenport, et al. 2017). In these plots we also include the VirtualBest solver, an artificial solver that always selects the best-performing algorithm for each instance. The legends are sorted with algorithms that solve more instances (using their average runtime as a tiebreaker) appearing higher. In addition, we list two values in the legend, namely the number of solved instances and the percentage contribution to VirtualBest. The higher this percentage is, the more instances can be solved best with this method.

All experiments were run on an *Intel Xeon E5-2623 v3* processor with *128* GB of RAM under Debian *10*.

6.2 Random Benchmark Suites

This section covers our random benchmarks. On the one hand, we look at instances coming from random XNF formulae and, on the other hand, from random systems of quadratic Boolean polynomials.

Benchmark 6.3 (*Random System of Linear Equations with 2-XNF Constraints*)**.** First we consider *sparse* random 2-XNF instances complemented with a *dense* system of linear equations, i.e., unit XNF clauses. More precisely, we generate a system of $\lfloor \frac{n}{2} \rfloor$ random linear equations in $\mathbb{L}_n$ along with n random 2-XNF clauses whose linerals ℓ satisfy $\#\operatorname{Supp}(\ell) \leq \frac{n}{20}$. Analogous to the above, we ensure that the clauses have a satisfying assignment. Our benchmark set contains one instance for each $n \in \{61, \ldots, 110\}$.

The cactus plot in Figure 3 shows a large gap between modern CNF-XOR- and XNF-based solvers on one side and logical CNF-based ones on the other. The reason for this separation can be found in the inability of the latter to handle larger systems of (dense) linear equations. Both variants of Bosphorus, combining logical with algebraic reasoning, land in the middle. Surprisingly, however, CryptoMiniSat 2.4 exhibited the worst performance, just behind Kissat and Glucose.

Benchmark 6.4 (*Multivariate Quadratic (MQ) Problems*)**.** Next we look at systems of random quadratic Boolean polynomials. In *multivariate cryptography*, the security of encryptions and signatures are based on the hardness of solving such random systems, see (Courtois, Goubin, et al. 2002; Kipnis et al. 1999). For each $n \in \{11, \ldots, 35\}$, we consider two systems of m random quadratic polynomials in $\mathbb{B}_n$. Similar to *Fukuoka's MQ-Challenge* (Yasuda et al. 2015), we consider different choices for m. Specifically, we look at two types: *Type I* instances satisfy $m = 2n$ and have at least one solution (which is enforced by fixing a random solution and flipping the constants of the polynomials appropriately), and *Type IV* instances satisfy $n = \lfloor 1.5 \cdot m \rfloor$. Clearly, the original problem encoding is in ANF. The other formats are computed as described in Remark 6.2.

Figure 4.a shows the cactus plot for the *Type I* systems. These are overdetermined, and PolyBoRi has the best performance on them. This might be attributed to the fact that the ANF representations are quite compact with only m polynomials and $n \leq 35$ variables. The graph-based solver 2-Xornado follows closely behind, as does CryptoMiniSat `-ext`. Except for Bosphorus-C, all the remaining solvers follow with quite similar performance,

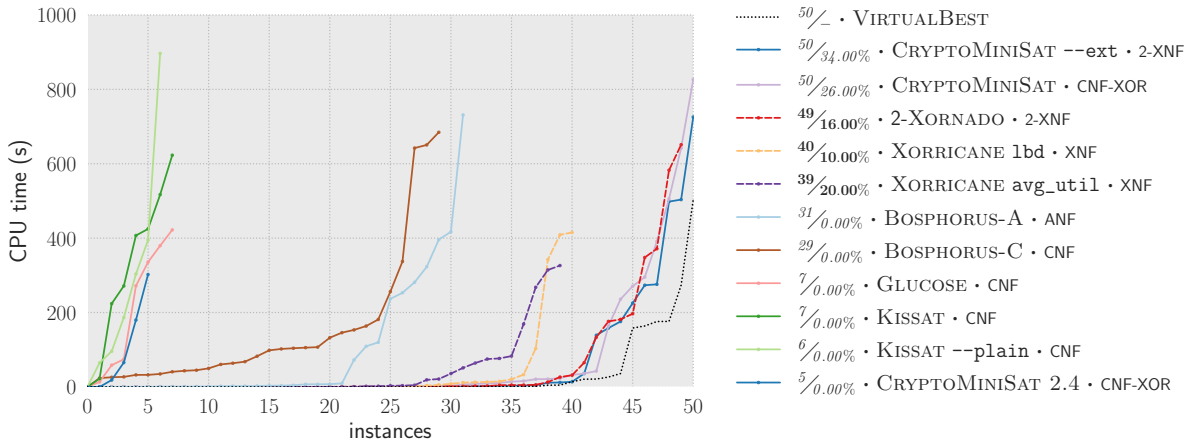


Fig. 3. Cactus plot for Benchmark 6.3 consisting of random systems of linear equations in $\mathbb{L}_n$ of size $\lfloor \frac{n}{2} \rfloor$ with n 2-XNF clauses whose linerals contain at most $\frac{n}{20}$ variables, where $n \in \{61, \ldots, 110\}$.

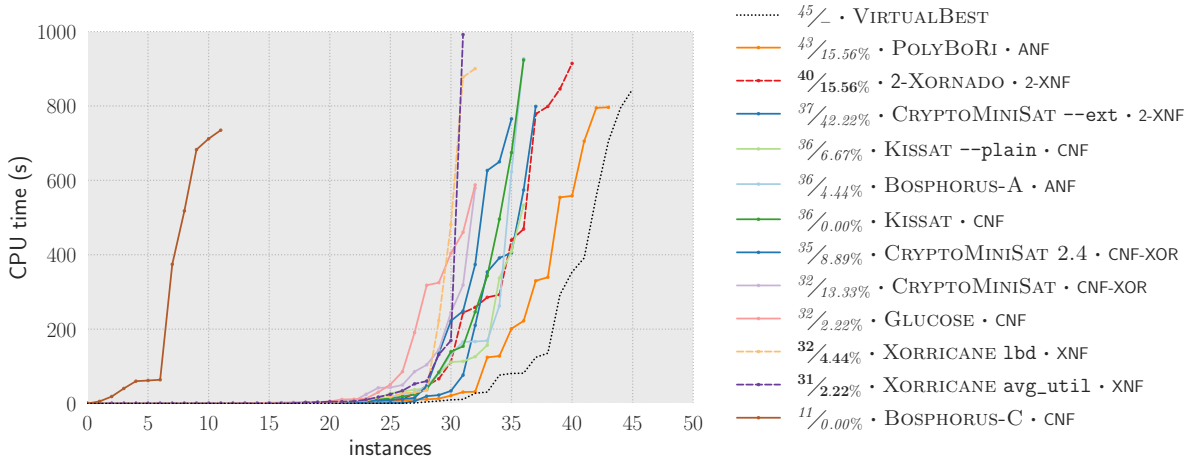
solving between 31 and 37 instances. In particular, Xorricane lbd solves as many instances as Glucose and CryptoMiniSat. Interestingly, Kissat –plain solves the exact same instances as Kissat but faster, i.e., the additional in- and preprocessing techniques in Kissat do not appear to help.

For the *Type IV* systems, see Figure 4.b, the plain version even outperforms Kissat, showing that these techniques are disadvantageous. Furthermore, PolyBoRi's algebraic reasoning suffers from many crashes due to excessive RAM usage, and it has one of the worst performances, leaving the top spot to the DPLL-based solver 2-Xornado. It is closely followed by the vast majority of the solvers, which solve between 28 and 33 instances. In particular, Xorricane lbd can solve as many instances as CryptoMiniSat 2.4, one less than CryptoMiniSat, and five more than Glucose.

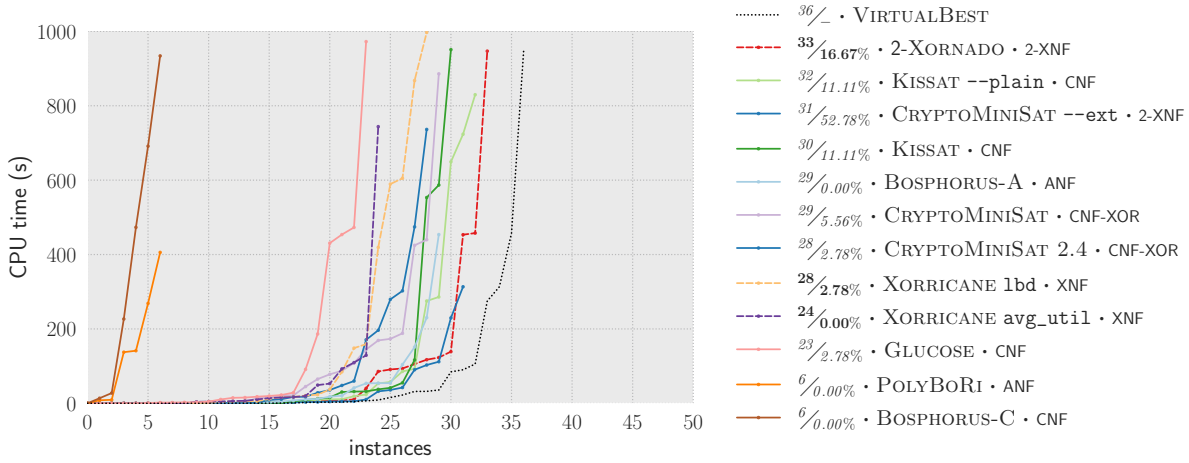
In contrast to Benchmark 6.3, which considered random XNF formulae, here random systems in ANF are considered. As a result, the corresponding XNF encodings are much more structured but contain auxiliary variables. For example, the largest instance solved by Xorricane lbd has 1107 clauses in 378 variables.

To conclude this section, Figure 5.a shows a cactus plot for all three random benchmark suites. 2-Xornado emerges as the best overall solver, just ahead of CryptoMiniSat –ext and closely followed by CryptoMiniSat. Next is Xorricane lbd, ahead of Bosphorus-A, Xorricane avg_util, and all other CNF-based methods.

To highlight the trade-off between propagation/learning power and the speed of the implementation, the cactus plot in Figure 5.b shows the same results for the logical solvers but with an additional limit of 10^6 on the number of decisions. Note that the percentage contribution of XNF-based methods to VirtualBest more than doubles. Moreover, a vast majority of 117 out of 123 instances were solved with the fewest decisions by 2-Xornado and Xorricane lbd. In particular, this shows that the propagation mechanism and decision heuristic of 2-Xornado are rather effective on these benchmarks. Furthermore, the number of instances solved by CryptoMiniSat and Kissat is significantly reduced, whereas both variants of Xorricane can solve the same instances. This emphasizes that propagation in our new XNF-based methods is much slower than in traditional CNF-based methods. However, this is (partly) compensated for by learning stronger clauses from conflicts and thus requiring fewer decisions.



(a) Cactus plot for the benchmark suite consisting of overdetermined systems of *Type I*.



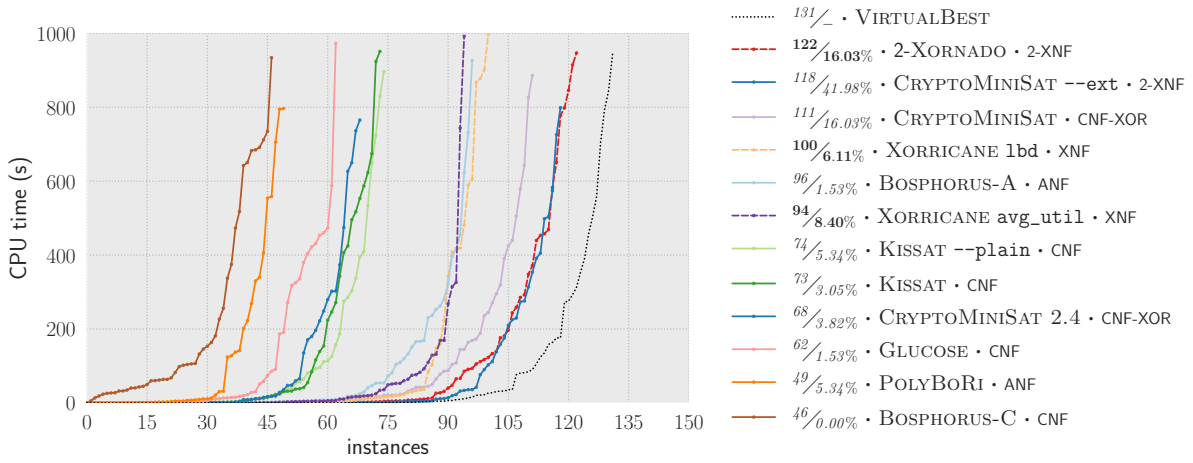
(b) Cactus plot for the benchmark suite consisting of underdetermined systems of *Type IV*.

Fig. 4. Cactus plot for Benchmark 6.4 consisting of systems of random multivariate quadratic polynomials with $n \in \{11, \ldots, 35\}$ indeterminates.

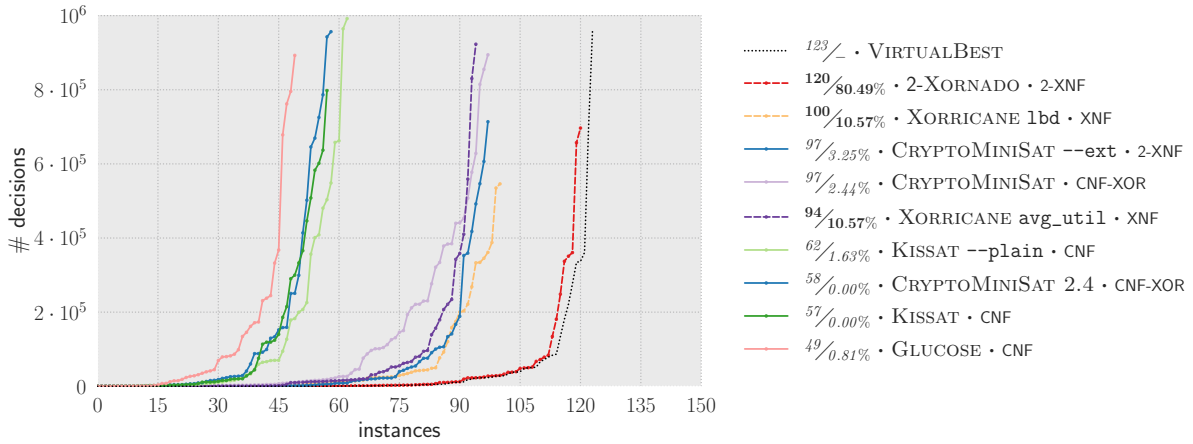
6.3 Cryptographic Benchmark Suites

This section considers structured XOR-rich instances in the context of cryptographic state- and key-recovery attacks on stream and block ciphers. In particular, we consider simplified round-reduced variants for attacks on three different cryptosystems of varying difficulty. Each benchmark suite contains 50 satisfiable instances. Unless otherwise stated, the problems are initially encoded in ANF before being translated into the other normal forms as described in Remark 6.2.

Benchmark 6.5 (*Ascon*)**.** Consider benchmarks related to key-recovery attacks on round-reduced versions of the block cipher Ascon-128 (Dobraunig et al. 2021). In particular, we consider attacks where the 128-bit nonce and



(a) Cactus plot of all solvers with a 1000 second timeout.



(b) Cactus plot of logical solvers with a 1000 second timeout and 10^6 decision limit.

Fig. 5. Overview of benchmark results for all 150 random instances.

the 320-bit internal state are known, and the goal is to *undo* the initialization step consisting of 12 rounds in order to obtain the 128-bit secret key. If this problem can be solved efficiently, the cipher is broken in a nonce-misuse scenario, see (Baudrin et al. 2022). Here we consider round-reduced variants: 10 instances with 2 rounds and no knowledge of key bits, 20 instances with 3 rounds and knowledge of the first $k \in \{45, \ldots, 64\}$ key bits, and 20 instances with 4 rounds and knowledge of the first $k \in \{81, \ldots, 100\}$ key bits. The instances were generated analogous to the ones in (Andraschko et al. 2024), i.e., the encodings were augmented with some additional XNF clauses to speed up propagation of 2-Xornado. These XNF encodings consist of up to 10340 clauses in 3136 variables.

It turns out that for the 2-round instances, 2-Xornado and Xorricane can solve all instances during preprocessing. For these instances, Kissat requires more than 16 000 decisions on average. Also, on 3- and 4-round

instances, 2-Xornado prevails. In the latter case, it even uses, on average, fewer decisions than Kissat by a factor of almost 1200. While Xorricane `avg_util` cannot solve as many instances, it still requires fewer decisions by a factor of almost 23 on the problems both methods could solve. Figure 6 shows the results for all instances, indicating 2-Xornado as the overall best solver. Furthermore, both variants of Xorricane can solve more instances than CryptoMiniSat, Kissat `-plain`, and Glucose.

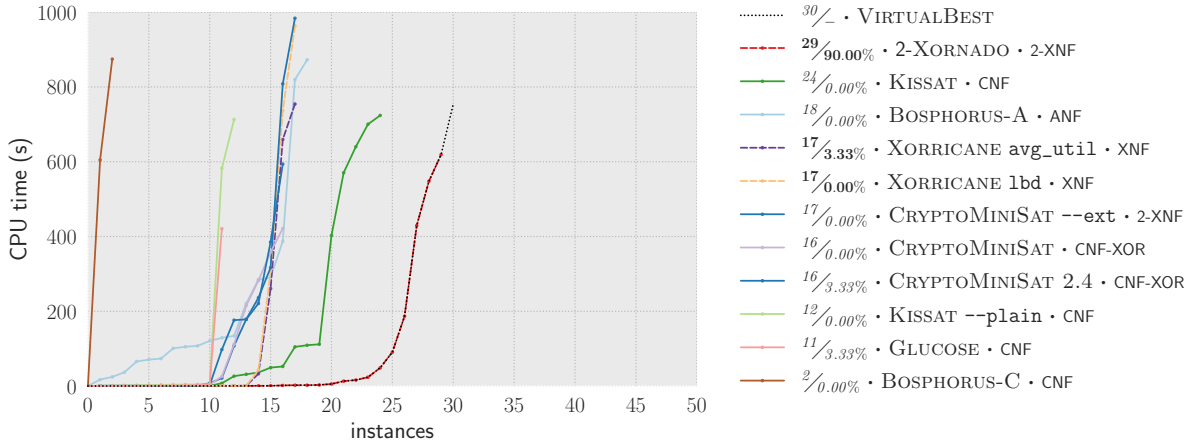


Fig. 6. Cactus plot for Benchmark 6.5 consisting of 50 satisfiable instances related to state-recovery attacks on Ascon-128.

Benchmark 6.6 (*Bivium*)**.** Trivium is a stream cipher of the *eSTREAM portfolio* (Robshaw and Billet 2008) and provides a good trade-off between security, speed and gate count in hardware implementations. It consists of three cyclically connected *shift registers* with non-linear output functions. In order to study its security, a simplified variant, Bivium, consisting of two of the three registers with a total size of 177 bits was introduced in (Maximov and Biryukov 2007). An 80-bit key and an 80-bit initialization vector are used to construct an *initial state*. Each *update* to this state creates a new output bit that can be used as a *pseudo-random source* for encryption. In a known-plaintext scenario, this output of the stream cipher is known. In this setting, we construct instances corresponding to state-recovery attacks. For each $k \in \{26, \ldots, 50\}$ and $r \in \{2 \cdot 177, 3 \cdot 177\}$, we consider an instance where the last k bits of the state and r consecutive bits of the output sequence are known. We use a straightforward 3-XNF encoding of the cipher to generate the instances.

The graph-based solver 2-Xornado struggles with these instances, see Figure 7. Given that the 2-XNFs are generally rather small (up to 3767 clauses in 1239 variables), this exemplifies how the size of a given encoding is a bad measure for problem complexity. However, with the exception of Bosphorus, all conflict-driven solvers benefit from the structure and solve at least 36 instances. In particular, our XNF-based solver Xorricane `avg_util` delivers the best performance, outperforming Kissat and both variants of CryptoMiniSat. It again requires fewer decisions than Kissat, but this time only by an average factor of about 1.5. In addition, CryptoMiniSat 2.4 provides the best timing for 23 instances, makes the largest contribution to VirtualBest, and performs better than CryptoMiniSat.

Benchmark 6.7 (*CTC2*)**.** The block cipher CTC2 is based on a substitution-permutation network and was introduced in (Courtois 2007a,b) as a tool to assess the potential of algebraic attacks on block ciphers. It consists of a *substitution* and a *linear diffusion* layer. The cipher has two parameters to tune: the number of rounds r and the block size b, which is the number of parallel *S-Boxes* in the substitution layer per round and determines the

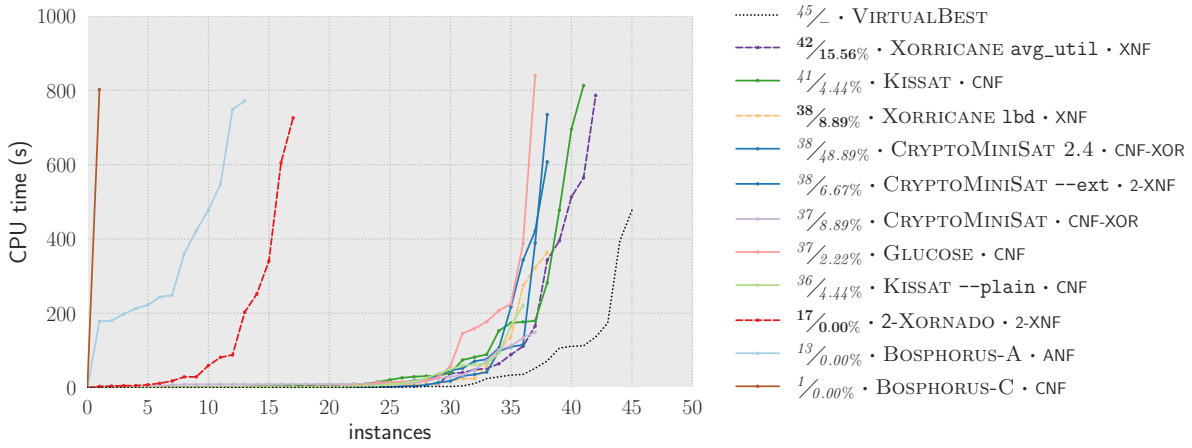


Fig. 7. Cactus plot for Benchmark 6.6 consisting of 50 satisfiable instances related to state-recovery attacks on the Bivium stream cipher.

key size as $k = 3b$. Our benchmark set contains instances related to key-recovery attacks on CTC2, where we have two instances for each $b \in \{9, \ldots, 13\}$ and $r \in \{2, \ldots, 6\}$, and where $m = \lfloor \frac{b+1}{2} \rfloor$ known plaintext-ciphertext pairs are known.

Figure 8 contains a cactus plot for this benchmark suite. The instances with $r \in \{5, 6\}$ proved the hardest, with only 4 out of 20 instances solved by any method. This time, Kissat is in the lead with the largest contribution to VirtualBest. It is closely followed by Bosphorus-A, which may have benefited from the more compact ANF input, unlike the previous benchmark results. It is also the first time PolyBoRi could solve any cryptographic instance within the timeout. Then we have Glucose and both variants of Xorricane, just ahead of the variants of CryptoMiniSat and Kissat `-plain`. In particular, our conflict-driven XNF SAT solvers are almost on par with Glucose and slightly better than CryptoMiniSat and Kissat `-plain`. At the lower end, we find 2-Xornado and PolyBoRi.

A cactus plot showing the performance of the solvers on all cryptographic instances is given in Figure 9.a. For 118 of these 150 instances, Kissat can find a solution within a 1000 second timeout, making it the best solver. In second place is CryptoMiniSat 2.4 from 2010, which solves 94 instances. It beats CryptoMiniSat by only three instances. However, in between is our new conflict-driven XNF SAT solver Xorricane `avg_util`, solving more instances than CryptoMiniSat, CryptoMiniSat `-ext`, Glucose, Kissat `-plain` and Bosphorus. Figure 9.b contains a cactus plot for all logical solvers, where in addition to the 1000 second timeout, a limit of 10^6 decisions is imposed. There we see that the XNF-based methods require far fewer decisions to terminate, indicating the strength of their propagation and clause learning procedures. Moreover, CryptoMiniSat generally performs better than CryptoMiniSat `-ext`, i.e., *enhancing* the propagation as outlined in Remark 5.6 does not pay off on these benchmarks.

7 Conclusion

In this paper, we have generalized CNF- and CNF-XOR-based conflict-driven SAT solving based on the resolution proof system RES to the setting of the more expressive XNF in CDXCL$^{\text{lite}}$. For its proof theoretic foundation,

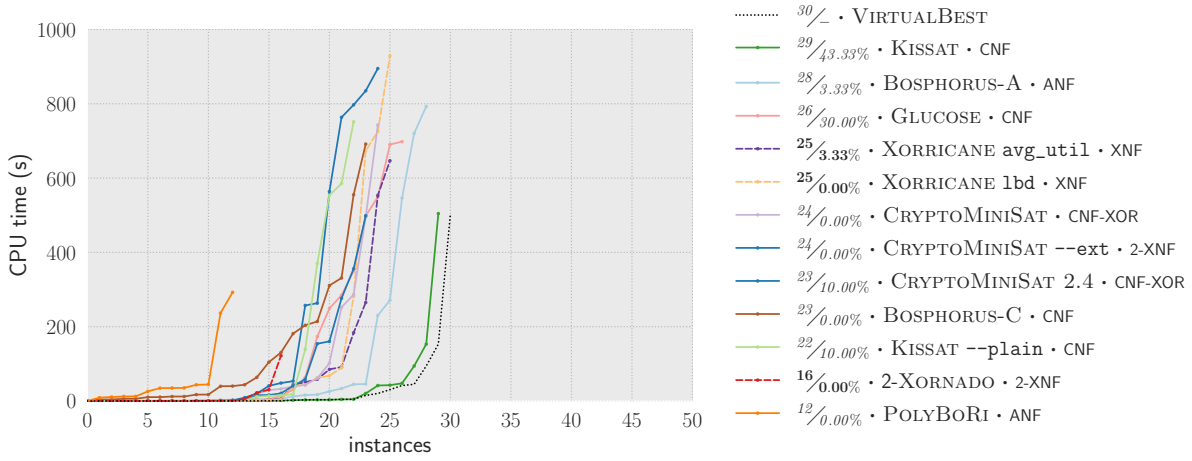


Fig. 8. Cactus plot for Benchmark 6.7 consisting of 50 satisfiable instances related to key-recovery attacks on different variants of the CTC block cipher.

it reasons using the proof system XLIN which has been proven to be exponentially stronger than RES. Subsequently, we introduced lazy data structures that allow an efficient implementation, as demonstrated by our solver Xorricane. Moreover, we made some first suggestions for inprocessing methods. Despite the fact that our implementation is still in its infancy, see Remark 6.1, it outperforms CryptoMiniSat, a modern CNF-XOR SAT solver, on cryptographic benchmark suites, and Kissat, a state-of-the-art CNF SAT solver, on random benchmark suites. This also indicates that XNF-based methods may have the potential to compete and outperform established CNF-XOR- and CNF-based methods if given equal attention and development.

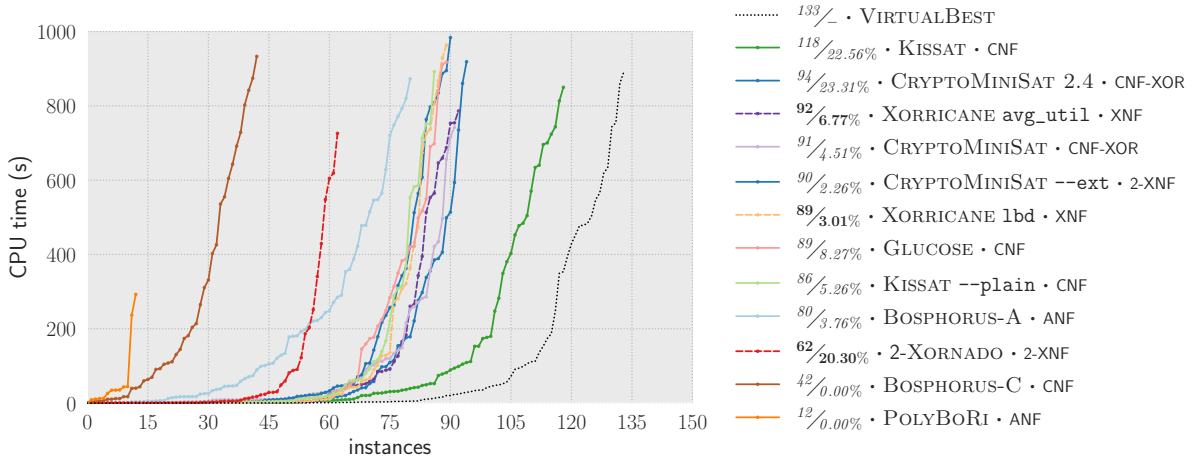
To conclude this paper, we highlight a few important aspects for efficient, practical XNF-based SAT solving. These should be addressed in future research.

Remark 7.1 (Implementation)**.** Undeniably, the XNF-based propagation and learning algorithms, GCP^{lite} and LinTrailRes, cannot be implemented as efficiently as their CNF counterparts. Although our implementation already uses custom lazy data structures, there is substantial room for improvement, as it does not take into account its cache usage. If the situation is similar to that of CNF-based solvers, an average speed-up of 60% might be possible by providing a resource-aware implementation, see (Chu et al. 2010; Hölldobler et al. 2010).

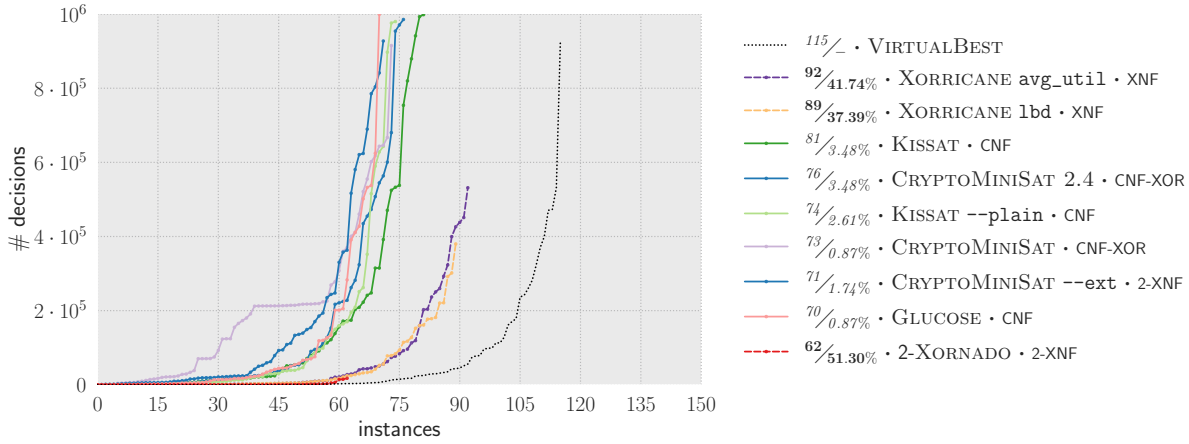
Remark 7.2 (Heuristics)**.** The parameters for the restart, decisions, and deletion heuristics of Xorricane were chosen from a small, reasonable set of values that performed best on a small benchmark suite with a timeout of only 120 seconds. Therefore, the long-term effects of these choices are unclear. In order to compete with established solvers, a systematic study of the selection and fine-tuning of these heuristics, similar to (Biere and Fröhlich 2018, 2015), using a broader set of benchmarks and a larger timeout is needed.

Remark 7.3 (In- and Preprocessing)**.** From an algorithmic point of view, Xorricane should also incorporate further key components of efficient conflict-driven SAT solving. Some are relatively straightforward to adapt, such as *clause minimization* (Beame, Kautz, et al. 2004; Eén and Sörensson 2004; Sörensson and Biere 2009) and *chronological backtracking* (Nadel and Ryvchin 2018). Other in- and preprocessing techniques, however, seem easy to generalize but hard to implement efficiently.

It is at hand to develop an approach similar to the BIRD framework (Soos, Gocht, et al. 2020; Soos and Meel 2019) used successfully in the context of CNF-XOR SAT solving. Translated to our setting, the idea of the framework is



(a) Cactus plot of all solvers with a 1000 second timeout.



(b) Cactus plot of logical solvers with a 1000 second timeout and 10^6 decision limit.

Fig. 9. Overview of benchmark results for all 150 cryptographic instances.

to encode the input XNF problem as a CNF formula to which established methods can be applied. Then, in a second step, new XNF information is extracted and added to the clause database of the XNF SAT solver.

Remark 7.4 (XNF Encodings)**.** Finally, XNF encodings should be studied more carefully. In particular, the encodings of Benchmark 6.6 (Bivium) come to mind. Here, Xorricane `avg_util` outperforms all other solvers when presented with the XNF encodings. However, given the 2-XNF encodings, it can solve only 34, i.e., almost 20% fewer instances. This shows that Xorricane is very sensitive to changes in the input encoding. While this can be partly compensated by new and better preprocessing tools, a proper theoretical analysis using some notion of *propagation completeness* for GCP^{lite} akin to (Brain, Hadarean, et al. 2016; Eén and Biere 2005; Sinz 2005) should be explored.

References

B. Andraschko, J. Danner, and M. Kreuzer. 2024. "SAT solving using XOR-OR-AND normal forms." *Math. Comput. Sci.*, 18, Article 20, 26 pages. doi:10.1007/s11786-024-00594-x.

G. Audemard and L. Simon. 2009. "Predicting learnt clauses quality in modern SAT solvers." In: *Proc. Int. Jt. Conference on Artificial Intelligence (IJCAI'09)*. Morgan Kaufmann Publishers Inc., Pasadena, California, USA, 399–404. https://www.ijcai.org/Proceedings/09/Papers/074.pdf.

T. Balyo, A. Fröhlich, M. Heule, and A. Biere. 2014. "Everything you always wanted to know about blocked sets (but were afraid to ask)." In: *Theory and Applications of Satisfiability Testing (SAT 2014)* (LNCS). Ed. by C. Sinz and U. Egly. Vol. 8561. Springer Int. Publ., Cham, 317–332. ISBN: 978-3-319-09284-3. doi:10.1007/978-3-319-09284-3_24.

G. Bard, N. Courtois, and C. Jefferson.. 2007. *Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over GF(2) via SAT-solvers.* Cryptology ePrint Archive, Paper 2007/024. https://eprint.iacr.org/2007/024. (2007). https://eprint.iacr.org/2007/024.

J. Baudrin, A. Canteaut, and L. Perrin. 2022. "Practical cube attack against nonce-misused Ascon." *IACR Trans. Symmetric Cryptol.*, 2022, 4, 120–144. doi:10.46586/tosc.v2022.i4.120-144.

P. Beame, H. Kautz, and A. Sabharwal. Dec. 2004. "Towards understanding and harnessing the potential of clause learning." *J. Artif. Int. Res.*, 22, 1, (Dec. 2004), 319–351. doi:10.1613/jair.1410.

P. Beame and S. Koroth. 2023. "On disperser/lifting properties of the index and inner-product functions." In: *Innovations in Theoretical Computer Science Conference (ITCS 2023)* (LIPIcs). Ed. by Y. Tauman Kalai. Vol. 251. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 14:1–14:17. ISBN: 978-3-95977-263-1. doi:10.4230/LIPIcs.ITCS.2023.14.

A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. 2020. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020." In: *Proc. SAT Competition 2020*. Department of Computer Science, University of Helsinki, Helsinki, 50–53. https://helda.helsinki.fi/handle/10138/318450.

A. Biere, M. Fleury, N. Froleyks, and M. Heule. 2023. "The SAT museum." In: *Proc. Pragmatics of SAT (PoS 2023)* (CEUR Workshop Proceedings). Ed. by M. Järvisalo and D. Le Berre. Vol. 3545. CEUR-WS.org, Alghero, Italy, 72–87. http://ceur-ws.org/Vol-3545.

A. Biere and A. Fröhlich. 2018. "Evaluating CDCL restart schemes." In: *Proc. Pragmatics of SAT 2015 and 2018* (EPiC Series in Computing). Ed. by D. Le Berre and M. Järvisalo. Vol. 59. EasyChair, Stockport, 1–17. doi:10.29007/89DW.

A. Biere and A. Fröhlich. 2015. "Evaluating CDCL variable scoring schemes." In: *Theory and Applications of Satisfiability Testing (SAT 2015)* (LNCS). Ed. by M. Heule and S. Weaver. Vol. 9340. Springer Int. Publ., Austin, TX, USA, 405–422. doi:10.1007/978-3-319-24318-4_29.

A. Biere, M. Heule, H. van Maaren, and T. Walsh, (Eds.) . 2021. *Handbook of Satisfiability*. (2nd ed.). Frontiers in Artificial Intelligence and Applications. Vol. 336. IOS Press, Amsterdam. ISBN: 978-1-64368-160-3. doi:10.3233/FAIA336.

M. Brain, J. Davenport, and A. Griggio. 2017. *Benchmarking solvers, SAT-style.* (2017).

M. Brain, L. Hadarean, D. Kroening, and R. Martins. 2016. "Automatic generation of propagation complete SAT encodings." In: *Verification, Model Checking, and Abstract Interpretation (VMCAI 2016)* (LNCS). Ed. by B. Jobstmann, K. Leino, and M. Rustan. Vol. 9583. Springer, Berlin, Heidelberg, 536–556. ISBN: 978-3-662-49122-5. doi:10.1007/978-3-662-49122-5_26.

M. Brickenstein and A. Dreyer. 2009. "PolyBoRi: a framework for Gröbner-basis computations with Boolean polynomials." *J. Symbolic Comput.*, 44, 9, 1326–1345. doi:10.1016/j.jsc.2008.02.017.

S. Bulygin and J. Buchmann. 2011. "Algebraic cryptanalysis of the round-reduced and side channel analysis of the full PRINTCipher-48." In: *Cryptology and Network Security (CANS 2011)* (LNCS). Ed. by D. Lin, G. Tsudik, and X. Wang. Vol. 7092. Springer, Berlin, Heidelberg, 54–75. ISBN: 978-3-642-25513-7. doi:10.1007/978-3-642-25513-7_6.

A. Chattopadhyay, N. Mande, S. Sanyal, and S. Sherif. 2023. "Lifting to parity decision trees via stifling." In: *Innovations in Theoretical Computer Science Conference (ITCS 2023)* (LIPIcs). Ed. by Y. Tauman Kalai. Vol. 251. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 33:1–33:20. ISBN: 978-3-95977-263-1. doi:10.4230/LIPIcs.ITCS.2023.33.

D. Choo, M. Soos, K. Chai, and K. Meel. 2019. "Bosphorus: bridging ANF and CNF solvers." In: *Design, Automation, and Test in Europe (DATE)*. IEEE Xplore, Florence 2019, 468–473. doi:10.23919/DATE.2019.8715061.

G. Chu, A. Harwood, and P. Stuckey. 2010. "Cache conscious data structures for Boolean satisfiability solvers." *J. Satisf. Boolean Model. Comput.*, 6, 99–120. 1-3. doi:10.3233/SAT190064.

N. Courtois. 2007a. *CTC2 and fast algebraic attacks on block ciphers revisited.* Cryptology ePrint Archive, Paper 2007/152. (2007). https://eprint.iacr.org/2007/152.

N. Courtois. 2007b. "How fast can be algebraic attacks on block ciphers?" In: *Symmetric Cryptography* (Dagstuhl Seminar Proceedings (DagSemProc)). Ed. by E. Biham, H. Handschuh, S. Lucks, and V. Rijmen. Vol. 7021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1–6. doi:10.4230/DagSemProc.07021.8.

N. Courtois, L. Goubin, W. Meier, and J.-D. Tacier. 2002. "Solving underdefined systems of multivariate quadratic equations." In: *Public Key Cryptography (PKC 2002)* (LNCS). Ed. by D. Naccache and P. Paillier. Vol. 2274. Springer, Berlin, Heidelberg, 211–227. ISBN: 978-3-540-45664-3. doi:10.1007/3-540-45664-3_15.

N. Courtois and J. Patarin. 2003. "About the XL algorithm over GF(2)." In: *Topics in Cryptology (CT-RSA 2003)* (LNCS). Ed. by M. Joye. Vol. 2612. Springer, San Francisco, CA, USA, 141–157. ISBN: 978-3-540-36563-1. doi:10.1007/3-540-36563-X_10.

N. Courtois, P. Sepehrdad, P. Sušil, and S. Vaudenay. 2012. "The ElimLin algorithm revisited." In: *Proc. Fast Software Encryption (FSE 2012)* (LNCS). Vol. 7549. Springer-Verlag, Washington, USA, 306–325. doi:10.1007/978-3-642-34047-5_18.

M. Davis, G. Logemann, and D. Loveland. July 1962. "A machine program for theorem proving." *Commun. ACM*, 5, 7, (July 1962), 394–397. doi:10.1145/368273.368557.

C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. July 2021. "Ascon v1.2: lightweight authenticated encryption and hashing." *J. Cryptol.*, 34, 3, (July 2021), 42 pages. doi:10.1007/s00145-021-09398-9.

A. Dreyer and T. Nguyen. 2013. "Improving Gröbner-based clause learning for SAT solving industrial-sized Boolean problems." In: *Proc. Young Researcher Symposium (YRS 2013)*. Fraunhofer Verlag, Stuttgart, 72–77. https://publica.fraunhofer.de/handle/publica/383437.

N. Eén and A. Biere. 2005. "Effective preprocessing in SAT through variable and clause elimination." In: *Theory and Applications of Satisfiability Testing (SAT 2005)*. Ed. by F. Bacchus and T. Walsh. Springer, Berlin, Heidelberg, 61–75. ISBN: 978-3-540-31679-4. doi:10.1007/11499107_5.

N. Eén and N. Sörensson. 2004. "An extensible SAT-solver." In: *Theory and Applications of Satisfiability Testing (SAT 2004)*. Ed. by E. Giunchiglia and A. Tacchella. Springer, Berlin, Heidelberg, 502–518. ISBN: 978-3-540-24605-3. doi:10.1007/978-3-540-24605-3_37.

K. Efremenko, M. Garlík, and D. Itsykson. 2024. "Lower bounds for regular resolution over parities." In: *Proc. ACM Symp. Theory of Computing (STOC 2024)*. ACM, Vancouver, BC, Canada, 640–651. ISBN: 9798400703836. doi:10.1145/3618260.3649652.

G. Emdin, A. Kulikov, I. Mihajlin, and N. Slezkin. 2024. "CNF encodings of symmetric functions." *Theory Comput. Syst.*, 68, 1291–1311. doi:10.1007/s00224-024-10168-w.

M. Garlík and L. Kołodziejczyk. Nov. 2018. "Some subsystems of constant-depth Frege with parity." *ACM Trans. Comput. Logic*, 19, 4, Article 29, (Nov. 2018), 34 pages. doi:10.1145/3243126.

I. Gent. Oct. 2013. "Optimal implementation of watched literals and more general techniques." *J. Artif. Int. Res.*, 48, 1, (Oct. 2013), 231–252. doi:10.1613/jair.4016.

C. Gomes, B. Selman, and H. Kautz. 1998. "Boosting combinatorial search through randomization." In: *National Conference Artificial Intelligence (AAAI-98)* (AAAI '98/IAAI '98). AAAI Press, Madison, Wisconsin, USA, 431–437. ISBN: 0262510987. https://dl.acm.org/doi/10.5555/295240.295710.

S. Gryaznov, S. Ovcharov, and A. Riazanov. Sept. 2024. "Resolution over linear equations: combinatorial games for tree-like size and space." *ACM Trans. Comput. Theory*, 16, 3, Article 15, (Sept. 2024), 15 pages. doi:10.1145/3675415.

M. Gwynne and O. Kullmann. 2014. "On SAT representations of XOR constraints." In: *Language and Automata Theory and Applications (LATA 2014)* (LNCS). Ed. by A.-H. Dediu, C. Martín-Vide, J.-L. Sierra-Rodríguez, and B. Truthe. Vol. 8370. Springer, Cham, 409–420. ISBN: 978-3-319-04921-2. doi:10.1007/978-3-319-04921-2_33.

A. Haken. 1985. "The intractability of resolution." *Theor. Comput. Sci.*, 39, 297–308. Third Conference on Foundations of Software Technology and Theoretical Computer Science. doi:10.1016/0304-3975(85)90144-6.

Y. Hamadi, S. Jabbour, and L. Saïs. 2009. "Learning for dynamic subsumption." In: *Int. Conf. Tools with Artificial Intelligence (ICTAI 2009)*. IEEE, 328–335. doi:10.1109/ICTAI.2009.22.

C.-S. Han and J.-H. Jiang. 2012. "When Boolean satisfiability meets Gaussian elimination in a simplex way." In: *Computer Aided Verification (CAV 2012)*. Ed. by P. Madhusudan and S. Seshia. Springer-Verlag, Berlin, Heidelberg, 410–426. ISBN: 978-3-642-31424-7. doi:10.1007/978-3-642-31424-7_31.

H. Han and F. Somenzi. 2007. "Alembic: an efficient algorithm for CNF preprocessing." In: *Proc. Design Automation Conference (DAC)* (DAC '07). ACM, San Diego, California, 582–587. ISBN: 9781595936271. doi:10.1145/1278480.1278628.

H. Han and F. Somenzi. 2009. "On-the-fly clause improvement." In: *Theory and Applications of Satisfiability Testing (SAT 2009)* (LNCS). Ed. by O. Kullmann. Vol. 5584. Springer, Berlin, Heidelberg, 209–222. ISBN: 978-3-642-02777-2. doi:10.1007/978-3-642-02777-2_21.

M. Heule, M. Järvisalo, and A. Biere. 2013. "Revisiting hyper binary resolution." In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2013)* (LNCS). Ed. by C. Gomes and M. Sellmann. Vol. 7874. Springer-Verlag, Berlin, 77–93. ISBN: 978-3-642-38171-3. doi:10.1007/978-3-642-38171-3_6.

S. Hölldobler, N. Manthey, and A. Saptawijaya. 2010. "Improving resource-unaware SAT solvers." In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2010)* (LNCS). Ed. by C. Fermüller and A. Voronkov. Vol. 6397. Springer, Berlin, Heidelberg, 519–534. ISBN: 978-3-642-16242-8. doi:10.1007/978-3-642-16242-8_37.

J. Horáček. 2020. "Algebraic and logic solving methods for cryptanalysis." Ph.D. Dissertation. Universität Passau, Universität Passau.

J. Horáček, J. Burchard, B. Becker, and M. Kreuzer. 2017. "Integrating algebraic and SAT solvers." In: *Mathematical Aspects of Computer and Information Sciences (MACIS 2017)* (LNCS). Ed. by J. Blömer, I. Kotsireas, T. Kutsia, and D. Simos. Vol. 10693. Springer Int. Publ., Cham, 147–162. ISBN: 978-3-319-72453-9. doi:10.1007/978-3-319-72453-9_11.

J. Horáček and M. Kreuzer. 2018. "Refutation of products of linear polynomials." In: *Proc. Satisfiability Checking and Symbolic Computation (SC^2)*. Oxford, 33–47. http://ceur-ws.org/Vol-2189/.

D. Itsykson and D. Sokolov. 2014. "Lower bounds for splittings by linear combinations." In: *Mathematical Foundations of Computer Science (MFCS 2014)* (LNCS). Ed. by E. Csuhaj-Varjú, M. Dietzfelbinger, and Z. Ésik. Vol. 8635. Springer, Berlin, Heidelberg, 372–383. ISBN: 978-3-662-44465-8. doi:10.1007/978-3-662-44465-8_32.

D. Itsykson and D. Sokolov. 2020. "Resolution over linear equations modulo two." *Ann. Pure Appl. Log.*, 171, 1, 102722. doi:10.1016/j.apal.2019.102722.

M. Järvisalo, A. Biere, and M. Heule. 2010. "Blocked clause elimination." In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010)* (LNCS). Ed. by J. Esparza and R. Majumdar. Vol. 6015. Springer, Berlin, Heidelberg, 129–144. ISBN: 978-3-642-12002-2. doi:10.1007/978-3-642-12002-2_10.

M. Järvisalo, M. Heule, and A. Biere. 2012. "Inprocessing rules." In: *Automated Reasoning (IJCAR 2012)* (LNCS). Ed. by M. Gramlich, D. Miller, and U. Sattler. Vol. 7364. Springer, Berlin, Heidelberg, 355–370. ISBN: 978-3-642-31365-3. doi:10.1007/978-3-642-31365-3_28.

E. Khaniki. July 2022. "On proof complexity of resolution over polynomial calculus." *ACM Trans. Comput. Logic*, 23, 3, Article 16, (July 2022), 24 pages. doi:10.1145/3506702.

A. Kipnis, J. Patarin, and L. Goubin. 1999. "Unbalanced oil and vinegar signature schemes." In: *Advances in Cryptology (EUROCRYPT 1999)* (LNCS). Ed. by J. Stern. Vol. 1592. Springer, Berlin, Heidelberg, 206–222. ISBN: 978-3-540-48910-8. doi:10.1007/3-540-48910-X_15.

J. Krajiŭek. 2019. *Proof Complexity*. Encyclopedia of Mathematics and its Applications. Cambridge University Press. doi:10.1017/9781108242066.

M. Kreuzer and L. Robbiano. 2000. *Computational Commutative Algebra 1*. Springer-Verlag, Berlin. doi:10.1007/978-3-540-70628-1.

T. Krüger, J.-H. Lorenz, and F. Wörz. Aug. 2022. "Too much information: why CDCL solvers need to forget learned clauses." *PLoS ONE*, 17, 8, (Aug. 2022), 1–28. doi:10.1371/journal.pone.0272967.

T. Laitinen, T. Junttila, and I. Niemelä. 2012a. "Classifying and propagating parity constraints." In: *Principles and Practice of Constraint Programming (CP 2012)* (LNCS). Ed. by M. Milano. Vol. 7514. Springer, Berlin, Heidelberg, 357–372. ISBN: 978-3-642-33558-7. doi:10.1007/978-3-642-33558-7_28.

T. Laitinen, T. Junttila, and I. Niemelä. 2012b. "Conflict-driven XOR-clause learning." In: *Theory and Applications of Satisfiability Testing (SAT 2012)*. Ed. by A. Cimatti and R. Sebastiani. Springer, Berlin, Heidelberg, 383–396. ISBN: 978-3-642-31612-8. doi:10.1007/978-3-642-31612-8_29.

T. Laitinen, T. Junttila, and I. Niemelä. Nov. 2011. "Equivalence class based parity reasoning with DPLL(XOR)." In: *Int. Conf. Tools with Artificial Intelligence (ICTAI 2011)*. IEEE, (Nov. 2011), 649–658. doi:10.1109/ICTAI.2011.103.

T. Laitinen, T. Junttila, and I. Niemelä. 2010. "Extending clause learning DPLL with parity reasoning." In: *European Conference on Artificial Intelligence (ECAI 2010)*. IOS Press, NLD, 21–26. ISBN: 9781607506058. doi:10.3233/978-1-60750-606-5-21.

T. Laitinen, T. Junttila, and I. Niemelä. 2012c. *Extending clause learning SAT solvers with complete parity reasoning (extended version)*. (2012). arXiv: 1207.0988 [cs.LO]. doi:10.48550/arXiv.1207.0988.

T. Laitinen, T. Junttila, and I. Niemelä. 2013. "Simulating parity reasoning." In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2013)* (LNCS). Ed. by K. McMillan, A. Middeldorp, and A. Voronkov. Vol. 8312. Springer, Berlin, Heidelberg, 568–583. doi:10.1007/978-3-642-45221-5_38.

M. Lewis, T. Schubert, and B. Becker. 2005. "Speedup techniques utilized in modern SAT solvers." In: *Theory and Applications of Satisfiability Testing (SAT 2005)*. Ed. by F. Bacchus and T. Walsh. Springer, Berlin, Heidelberg, 437–443. ISBN: 978-3-540-31679-4. doi:10.1007/11499107_36.

C. Li. 2000. "Integrating equivalency reasoning into Davis-Putnam procedure." In: *National Conference on Artificial Intelligence (AAAI-2000)*. AAAI Press, 291–296. ISBN: 0262511126.

C. Li, F. Xiao, M. Luo, F. Manyà, Z. Lü, and Y. Li. 2020. "Clause vivification by unit propagation in CDCL SAT solvers." *Artificial Intelligence*, 279, 103197. doi:10.1016/j.artint.2019.103197.

M. Luby, A. Sinclair, and D. Zuckerman. 1993. "Optimal Speedup of Las Vegas Algorithms." In: *Israel Symposium on Theory and Computing Systems*. IEEE, 128–133. doi:10.1109/ISTCS.1993.253477.

I. Lynce and J. Marques-Silva. 2003. "Probing-based preprocessing techniques for propositional satisfiability." In: *Int. Conf. Tools with Artificial Intelligence (ICTAI 2003)*. IEEE, USA, 105–110. ISBN: 0769520383. doi:10.1109/TAI.2003.1250177.

J. Marques-Silva and K. Sakallah. 1996. "GRASP - a new search algorithm for satisfiability." In: *IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD'96)*. IEEE, San Jose, California, USA, 220–227. ISBN: 0818675977. doi:10.1109/iccad.1996.569607.

A. Maximov and A. Biryukov. 2007. "Two trivial attacks on Trivium." In: *Selected Areas in Cryptography (SAC 2007)* (LNCS). Ed. by C. Adams, A. Miri, and M. Wiener. Vol. 4876. Springer, Seoul, Korea, 36–55. ISBN: 978-3-540-77360-3. doi:10.1007/978-3-540-77360-3_3.

M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. 2001. "Chaff: engineering an efficient SAT Solver." In: *Proc. Design Automation Conference (DAC'01)*. ACM, Las Vegas 2001, 530–535. doi:10.1145/378239.379017.

A. Nadel and V. Ryvchin. 2018. "Chronological backtracking." In: *Theory and Applications of Satisfiability Testing (SAT 2018)* (LNCS). Ed. by O. Beyersdorff and C. Wintersteiger. Vol. 10929. Springer Int. Publ., Oxford, UK, 111–121. ISBN: 978-3-319-94144-8. doi:10.1007/978-3-319-94144-8_7.

W. Nawrocki, Z. Liu, A. Fröhlich, M. Heule, and A. Biere. 2021. "XOR local search for Boolean Brent equations." In: *Theory and Applications of Satisfiability Testing (SAT 2021)* (LNCS). Ed. by C.-M. Li and F. Manyà. Vol. 12831. Springer Nature Switzerland, Cham, 417–435. ISBN: 978-3-030-80223-3. doi:10.1007/978-3-030-80223-3_29.

C. Oh. 2015. "Between SAT and UNSAT: the fundamental difference in CDCL SAT." In: *Theory and Applications of Satisfiability Testing (SAT 2015)* (LNCS). Ed. by M. Heule and S. Weaver. Vol. 9340. Springer Int. Publ., Austin, TX, USA, 307–323. ISBN: 978-3-319-24318-4. doi:10.1007/978-3-319-24318-4_23.

V. Oparin. 2016. "Tight upper bound on splitting by linear combinations for pigeonhole principle." In: *Theory and Applications of Satisfiability Testing (SAT 2016)* (LNCS). Ed. by N. Creignou and D. Le Berre. Vol. 9710. Springer Int. Publ., Cham, 77–84. ISBN: 978-3-319-40970-2. doi:10.1007/978-3-319-40970-2_6.

K. Pipatsrisawat and A. Darwiche. 2007. "A lightweight component caching scheme for satisfiability solvers." In: *Theory and Applications of Satisfiability Testing (SAT 2007)*. Ed. by J. Marques-Silva and K. Sakallah. Springer, Berlin, Heidelberg, 294–299. ISBN: 978-3-540-72788-0. doi:10.1007/978-3-540-72788-0_28.

A. A. Razborov. 2004. "Resolution lower bounds for perfect matching principles." *J. Comput. Syst. Sci.*, 69, 1, 3–27. Special Issue on Computational Complexity 2002. doi:10.1016/j.jcss.2004.01.004.

M. Robshaw and O. Billet, (Eds.) . 2008. *New Stream Cipher Designs: The eSTREAM Finalists*. LNCS. Vol. 4986. Springer-Verlag, Berlin, Heidelberg. ISBN: 9783540683506. doi:10.1007/978-3-540-68351-3.

B. Selman, H. Kautz, and D. McAllester. 1997. "Ten challenges in propositional reasoning and search." In: *Proc. Int. Jt. Conference on Artificial Intelligence (IJCAI'97)*. Vol. 1. Morgan Kaufmann Publishers Inc., Nagoya, Japan, 50–54. http://ijcai.org/Proceedings/97-1/Papers/008.pdf.

C. Sinz. 2005. "Towards an optimal CNF encoding of Boolean cardinality constraints." In: *Principles and Practice of Constraint Programming (CP 2005)* (LNCS). Ed. by P. van Beek. Vol. 3709. Springer, Berlin, Heidelberg, 827–831. ISBN: 978-3-540-32050-0. doi:10.1007/11564751_73.

M. Soos. 2010. "Enhanced Gaussian elimination in DPLL-based SAT solvers." In: *POS-10. Pragmatics of SAT* (EPiC Series in Computing). Ed. by D. Le Berre. Vol. 8. EasyChair, Stockport, 2–14. doi:10.29007/G7SS.

M. Soos. 2023. *The inprocessing API of CryptoMiniSat*. accessed on 12.06.2024. (2023). https://www.msoos.org/2023/10/the-inprocessing-api-of-cryptominisat/.

M. Soos, S. Gocht, and K. Meel. 2020. "Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling." In: *Computer Aided Verification (CAV 2020)* (LNCS). Vol. 12224. Springer-Verlag, Los Angeles, CA, USA, 463–484. ISBN: 978-3-030-53287-1. doi:10.1007/978-3-030-53288-8_22.

M. Soos and K. Meel. 2019. "BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting." In: *Proc. Conference on Artificial Intelligence (AAAI-19)*. Vol. 33. AAAI Press, Palo Alto 2019, 1592–1599. doi:10.1609/aaai.v33i01.33011592.

M. Soos, K. Nohl, and C. Castelluccia. 2009. "Extending SAT solvers to cryptographic problems." In: *Theory and Applications of Satisfiability Testing (SAT 2009)* (LNCS). Ed. by O. Kullmann. Vol. 5584. Springer-Verlag, Berlin, 244–257. doi:10.1007/978-3-642-02777-2_24.

N. Sörensson and A. Biere. 2009. "Minimizing learned clauses." In: *Theory and Applications of Satisfiability Testing (SAT 2009)*. Ed. by O. Kullmann. Springer, Berlin, Heidelberg, 237–243. ISBN: 978-3-642-02777-2. doi:10.1007/978-3-642-02777-2_23.

J. Warners and H. van Maaren. 1998. "A two-phase algorithm for solving a class of hard satisfiability problems." *Operations Research Letters*, 23, 3, 81–88. doi:10.1016/S0167-6377(98)00052-2.

T. Yasuda, X. Dahan, Y.-J. Huang, T. Takagi, and K. Sakurai. Nov. 2015. "A multivariate quadratic challenge toward post-quantum generation cryptography." *ACM Commun. Comput. Algebra*, 49, 3, (Nov. 2015), 105–107. doi:10.1145/2850449.2850462.

C. Zengler and W. Küchlin. 2010. "Extending clause learning of SAT solvers with Boolean Gröbner bases." In: *Computer Algebra in Scientific Computing (CASC 2010)* (LNCS). Ed. by V. Gerdt, W. Koepf, E. Mayr, and E. Vorozhtsov. Vol. 6244. Springer, Berlin, Heidelberg, 293–302. ISBN: 978-3-642-15274-0. doi:10.1007/978-3-642-15274-0_26.

A Xorricane – A Conflict-Driven XNF SAT Solver

Algorithm CDXCL$^{\text{lite}}$ has been implemented in C++ as a solver called Xorricane. It uses the data structures and most optimizations introduced in Section 5.2. The source code is available at https://github.com/j-danner/Xorricane-paper. It reads XNF instances in the DIMACS-compatible format described in (Andraschko et al. 2024).

In the following we provide some details about the inner workings of Xorricane, namely its decision, deletion, and restart heuristics, as well as its options for in- and preprocessing.

`-decision-heuristic [vsids|lwl|swl|lex]`

As usual in modern SAT solvers, Xorricane uses decision heuristics which consist of a variable heuristic with *phase saving* (Remark 5.17.a). For the variable heuristic, the default choice is `vsids` which implements an *Exponential Variable State Independent Decaying Sum* heuristic, as described in (Biere and Fröhlich 2015). Hence, an activity score for each variable is stored, and those with the highest score are chosen as decisions. The scores are *bumped* whenever a variable occurs in a reason clause which is used during conflict analysis. Additionally, the variables which are *resolved on* during LinTrailRes* are bumped another time. Initially,

the bump value is $b = 1$. However, it is scaled after every conflict by $g = 1.01$. After 10^5 conflicts, the scale factor linearly increases until it reaches $g = 1.1$ at 10^6 conflicts.

Alternatively, one can use the variable heuristic `lex`, which chooses the lexicographically smallest unassigned variable, or `lwl` and `swl`, which pick the variable with the longest or respectively shortest watch list.

`-restart-heuristic [no|luby|lbd]`

Restarts can generally be deactivated with `no`, or be scheduled according to a *luby series* with a *base interval size* of 2^7 conflicts, see (Biere and Fröhlich 2018, Section 3.2) and (Luby et al. 1993).

The default option is `lbd`, a dynamic heuristic based on exponential moving averages (EMA) of the LBD values of newly learned clauses. It uses the scheme as introduced in (Biere and Fröhlich 2018, Section 4.2) with smoothing parameters $\alpha_s = 2^{-12}$ for the slow EMA, $\alpha_f = 2^{-5}$ for the fast EMA, and $\alpha' = 2^{-11}$ for the blocking EMA; and initially uses margin ratios of $c = 1.15$ for forcing restarts and $c' = 1.4$ for blocking restarts. After $5 \cdot 10^4$ conflicts, the margin ratio for blocking restarts is linearly increased to $c' = 1.55$ at 10^5 conflicts.

`-deletion-heuristic [avg_util|lbd]`

Whenever the solver schedules a restart, it deletes parts of the learned clauses in Γ to speed up propagation. To decide which clauses to delete, the set Γ is partitioned into three tiers, as described in (Oh 2015). New clauses C are added to $\Gamma_{\texttt{tier1}}$ if $\mathrm{LBD}(C) \leq 3$, to $\Gamma_{\texttt{tier2}}$ if $3 < \mathrm{LBD}(C) \leq 6$, and otherwise to $\Gamma_{\texttt{tier3}}$. Now the deletion heuristics decide which clauses are deleted in $\Gamma_{\texttt{tier3}}$, and which are moved from $\Gamma_{\texttt{tier2}}$ to $\Gamma_{\texttt{tier3}}$.

One way to assess the *value* of a clause is to track its *utility*. Initially, it is set to 0, but whenever a clause occurs as part of a reason clause used during conflict analysis or becomes a unit under an assignment during propagation, its *utility* is increased by one. Option `avg_util` computes the average utilities $a_2, a_3 \in \mathbb{Q}$ for tiers 2 and 3 separately. Then, all non-unit clauses from $\Gamma_{\texttt{tier3}}$ with a utility below $\frac{3}{4} \cdot a_3$ are removed, and non-unit clauses from $\Gamma_{\texttt{tier2}}$, whose utility is below $\frac{a_2}{8}$, are moved to $\Gamma_{\texttt{tier3}}$.

Alternatively, the *usefulness* of a clause can be measured efficiently using the lineral block distance (see Definition 4.43 and Remark 5.17.b). The LBD is computed for each clause once it becomes a unit under the current assignment, and is recomputed whenever the clause is used as part of a reason clause during conflict analysis. Now, option `lbd` first computes the distribution of the LBD values in tiers 2 and 3 separately. Based on this, the integer l_2 is computed such that $\frac{17}{18}$ of the tier 2 clauses have an LBD lower or equal to l_2. Similarly, the integer l_3 is computed such that $\frac{7}{16}$ of the tier 3 clauses have an LBD lower or equal to l_3. Then, all non-unit clauses from $\Gamma_{\texttt{tier3}}$ with an LBD above l_3 are removed, and non-unit clauses from $\Gamma_{\texttt{tier2}}$, whose LBD is above l_2, are moved to $\Gamma_{\texttt{tier3}}$.

`-no-lazy-gauss-jordan-elim`

If this flag is given, the solver uses $\mathrm{GCP}^{\mathrm{lite}}$ for propagation. Otherwise, it uses a more powerful propagation mechanism, whose implementation uses parts of the highly optimized code of CryptoMiniSat, see (Soos 2010; Soos, Gocht, et al. 2020). Unfortunately, as certain preconditions are not met in the context of CDXCL$^{\mathrm{lite}}$, it misses some implied literals and serves as an *incomplete* implementation of $\mathrm{GCP}^{\mathrm{lite}}_{\mathrm{GJE}}$.

The implementation of the propagation method processes assignments using a priority queue. Newly derived linerals are added to the queue in the following order: (1) conflict linerals, (2) linerals derived from *Gauß-Jordan eliminiation* or from *preprocessing*, (3) linerals known to reduce to a literal, (4) equivalence linerals, and (5) remaining linerals. Linerals are processed in the order of this queue during the main propagation loop. If a lineral corresponds to an implied literal, the corresponding variable assignment is propagated to the clauses and linerals, potentially adding new elements to the queue.

`-no-equiv-sub`
By default, the solver uses *equivalence linerals* to heuristically reduce implied linerals (see Remark 5.7.b). The flag deactivates this heuristic.

`-impl-lits [INT]`
Activate inprocessing as suggested in Remark 5.7.a, i.e., explicitly compute the set $\mathrm{ImpLits}(L_d)$ of implied literals using Proposition 5.2.b. If an integer $s > 0$ is provided, this set is computed after every s-th call to $\mathrm{GCP}^{\mathrm{lite}}$. Also, whenever a new literal is derived, we re-run $\mathrm{GCP}^{\mathrm{lite}}$. By default, it is deactivated, i.e., $s = 0$.

Additionally, if $s = -1$ is given, the scheduling of inprocessing is dynamically adapted as follows. Initially, it uses $s = 2^9$ and whenever the average number of found implied literals increases, s is increased by 8; otherwise, decreased to $\lfloor \frac{s}{2} \rfloor$.

`-preprocess [no|scc|fls_scc]`
For preprocessing, the solver collects all 2-XNF clauses in the given formula and applies the preprocessor of the graph-based 2-XNF solver Xorricane (Andraschko et al. 2024) with the given options to these clauses. All newly derived linerals are added as unit XNF clauses.

Additionally, whenever the solver operates in decision level 0 and new linerals have been derived, the above preprocessing is repeated another time. There is only one exception: `fls_scc` is downgraded to `scc` after 100 applications.

The parameter values for the heuristics have been selected by testing a handful of reasonable values on a small benchmark set of 15 instances and with a timeout of 120 seconds. Thus, it is strongly recommended that these heuristics are properly fine-tuned using a more extensive benchmark set and a significantly higher timeout to assess the long-term effects of the choices. Further shortcomings of our implementation in comparison to state-of-the-art CNF-based SAT solvers are listed in Remark 6.1.

B Enhancing $\mathrm{GCP}^{\mathrm{lite}}_{\mathrm{GJE}}$ Using CNF-XOR Encodings

One key difference of GCP and $\mathrm{GCP}^{\mathrm{lite}}_{\mathrm{GJE}}$ when propagating variable assignments $A \subseteq \mathbb{X}_n$ is the following: the first essentially performs a *Gauß-Jordan reduction* for all linerals which have a reason clause under A, whereas the latter only uses the linerals which are unit XNF clauses, i.e., reason clauses under $\emptyset$, for reduction.

To leverage this enhanced propagation in $\mathrm{GCP}^{\mathrm{lite}}_{\mathrm{GJE}}$, we encode each lineral in the XNF formula F with an auxiliary variable. This results in a CNF-XOR encoding of F. If done correctly, this brings the reasoning of GCP down to the level of unit clauses and allows $\mathrm{GCP}^{\mathrm{lite}}_{\mathrm{GJE}}$ to effectively *simulate* GCP for variable assignments.

For this, let $\mathbb{B}_{n,m}$ be the Boolean polynomial ring in the indeterminates $x_1, \ldots, x_n, y_1, \ldots, y_m$, let $\mathbb{X}_{n,m}$ be the set of assigning Boolean polynomials of $\mathbb{B}_{n,m}$, and let $\mathbb{L}_{n,m}$ be the set of linear Boolean polynomials of $\mathbb{B}_{n,m}$. Then we consider the following encodings.

Definition B.1. Let $F = \{C_1, \ldots, C_k\} \subseteq \mathcal{P}(\mathbb{L}_n)$ be an XNF formula.

(a) Let $L = \bigcup_{C \in F} C$ be the set of all linerals which occur in F. Let $m = \#L$ and $\{i_\ell \mid \ell \in L\} = \{1, \ldots, m\}$. The CNF-XOR formula $G \subseteq \mathcal{P}(\mathbb{L}_{n,m})$ with

(1) unit XNF clauses $\{y_{i_\ell} + \ell\}$ for all $\ell \in L$ and

(2) CNF clauses $C'_i = \{y_{i_\ell} \mid \ell \in C_i\}$ for $i \in \{1, \ldots, k\}$

is called the **trivial CNF-XOR encoding** of F.

(b) Let $F' = \{C' \subseteq \mathbb{L}_n \text{ reduced} \mid \text{there is } C \in F \text{ with } C' \equiv C\}$ be the XNF formula which contains all reduced clauses that are equivalent to a clause from F. Then we call the trivial CNF-XOR encoding of F' the **extended CNF-XOR encoding** of F.

Remark B.2. Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be an XNF formula. The equivalence class of a k-XNF clause $C \in F$ contains $O(2^k)$ many elements by Remark 2.17.a. Thus, a *direct* computation of the extended CNF-XOR encoding leads to an exponential increase in the number of clauses.

This can be circumvented by using a polynomial-size reduction to a 2-XNF formula G first, see (Andraschko et al. 2024, Algorithm 1). Then, the equivalence classes of the clauses of G contain at most three elements, and an extended CNF-XOR encoding of G can be found in polynomial time.

Let us now see how extended encodings can enhance the propagation power of $\mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}$.

Lemma B.3. Let F be an XNF formula, let $G \subseteq \mathcal{P}(\mathbb{L}_n)$ be the trivial CNF-XOR encoding of F, and let $A \subseteq \mathbb{X}_n$ be a valid variable assignment.

If $[\ell_1, \dots, \ell_k]$ is a lite lineral trail for G under A, then we have

$$\mathrm{ImpLits}\big(A \cup \{\ell_1, \dots, \ell_k\}\big) \;\subseteq\; \mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}(G, A).$$

Proof. Let $\ell \in \mathrm{ImpLits}(A \cup \{\ell_1, \dots, \ell_k\})$ and write $A' = (A \cup \{\ell_1, \dots, \ell_k\}) \cap \mathbb{X}_n$. By Definition 5.1, there are $\ell_{i_1}, \dots, \ell_{i_t} \in \{\ell_1, \dots, \ell_k\} \setminus \mathbb{X}_n$ with $\ell = (\ell_{i_1} + \cdots + \ell_{i_t})|_{A'}$. As $[\ell_1, \dots, \ell_k]$ is a lite lineral trail, we have $\ell_{i_1}, \dots, \ell_{i_t} \in \mathrm{GCP}^{\mathsf{lite}}(G, A) \subseteq \mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}(G, A)$. Let G_{unit} be the set of linerals corresponding to the unit clauses of G. All non-unit clauses are in CNF, and if one of them is a unit clause under a variable assignment, then its implied lineral is clearly a literal. Thus, $\ell_{i_j} \notin \mathbb{X}_n$ implies that its reason clause corresponds to a lineral in G_{unit}, i.e., $\ell_{i_j} \in G_{\mathsf{unit}}$. Hence, there is some $\ell' \in \langle G_{\mathsf{unit}} \rangle_{\mathbb{F}_2}$ with $\ell'|_{A'} = \ell$. By definition of $\mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}$, this shows $\ell \in \mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}(G, A)$. □

Proposition B.4. Let $F \subseteq \mathcal{P}(\mathbb{L}_n)$ be an XNF formula, $G \subseteq \mathcal{P}(\mathbb{L}_{n,m})$ be the extended CNF-XOR encoding of F, and $A \subseteq \mathbb{X}_n$ be a valid variable assignment. Then we have

$$\mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}(G, A) \cap \mathbb{X}_n \;=\; \mathrm{ImpLits}\big(\,\mathrm{GCP}_\sigma(F, A)\,\big).$$

Proof. Let τ be a term ordering of $\mathbb{B}_{n,m}$ that extends σ such that $y_j > x_i$ for all indices $j \in \{1, \dots, m\}$ and $i \in \{1, \dots, n\}$. Using the notation of Definition B.1 we let $G_{\mathsf{unit}} = \big\{y_{i_\ell} + \ell \mid \ell \in \bigcup_{C \in F'} C\big\}$ correspond to the unit XNF clauses of G. Clearly, we have $G_{\mathsf{unit}} \subseteq \mathrm{GCP}_\tau(G, A)$. Moreover, G_{unit} is a τ-assignment and by construction of G there is, for each $C \in F$, exactly one $C' \in G$ with $C'|_{G_{\mathsf{unit}}} = C$. Hence, the non-trivial clauses of $G|_{G_{\mathsf{unit}}}$ are exactly the XNF clauses of $F \subseteq \mathcal{P}(\mathbb{L}_n)$. With $A \subseteq \mathbb{X}_n$ we get

$$\begin{aligned}\langle \mathrm{GCP}_\tau(G, A)\rangle_{\mathbb{F}_2} \;&=\; \langle \mathrm{GCP}_\tau\big(G|_{G_{\mathsf{unit}}},\, A \cup G_{\mathsf{unit}}\big)\rangle_{\mathbb{F}_2} \;=\; \langle \mathrm{GCP}_\tau\big(F,\, A \cup G_{\mathsf{unit}}\big)\rangle_{\mathbb{F}_2} \\ &=\; \langle \mathrm{GCP}_\sigma(F, A) \cup G_{\mathsf{unit}}\rangle_{\mathbb{F}_2},\end{aligned}$$

i.e., we have $\langle \mathrm{GCP}_\sigma(F, A)\rangle_{\mathbb{F}_2} = \langle \mathrm{GCP}_\tau(G, A)\rangle_{\mathbb{F}_2} \cap \mathbb{L}_n$. In view of Remark 5.5 it now suffices to show

$$\mathrm{ImpLits}(\mathrm{GCP}_\sigma(G, A)) \;\subseteq\; \mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}(G, A).$$

In order to apply Lemma B.3, we write $\mathrm{GCP}_\sigma(G, A) = A \cup \{\ell_1, \dots, \ell_k\}$, where $[\ell_1, \dots, \ell_k]$ is a lineral trail for G under A. Next, let us show by induction on k that there is a lite lineral trail $[\ell'_1, \dots, \ell'_s]$ for G under A with $\langle A \cup \{\ell_1, \dots, \ell_k\}\rangle_{\mathbb{F}_2} \subseteq \langle A \cup \{\ell'_1, \dots, \ell'_s\}\rangle_{\mathbb{F}_2}$. The case $k = 0$ is clear. So let $k > 0$ and $[\ell'_1, \dots, \ell'_s]$ be a lite lineral trail for G under A with $\langle A \cup \{\ell_1, \dots, \ell_{k-1}\}\rangle_{\mathbb{F}_2} \subseteq \langle A \cup \{\ell'_1, \dots, \ell'_s\}\rangle_{\mathbb{F}_2}$.

Let $L = A \cup \{\ell'_1, \dots, \ell'_s\}$ and $C \in G$ be the reason clause for ℓ_k under $A \cup \{\ell_1, \dots, \ell_{k-1}\}$. If C is a unit XNF clause, then there is $\ell \in \bigcup F'$ with $C = \{y_{i_\ell} + \ell\}$. Define $\ell' \in \mathbb{L}_{n,m}$ uniquely by $\{\ell'\} \equiv C|_{L \cap \mathbb{X}_{n,m}}$. Then $[\ell'_1, \dots, \ell'_s, \ell']$ is a lite lineral trail for G under A. Moreover, $y_{i_\ell} + \ell \in \langle A \cup \{\ell'_1, \dots, \ell'_s, \ell'\}\rangle_{\mathbb{F}_2}$ and the induction hypothesis shows

$$\begin{aligned}\ell_k \;=\; (y_{i_\ell} + \ell)|_{A \cup \{\ell_1, \dots, \ell_{k-1}\}} \;&\in\; (y_{i_\ell} + \ell) \,+\, \langle A \cup \{\ell_1, \dots, \ell_{k-1}\}\rangle_{\mathbb{F}_2} \\ &\subseteq\; \langle A \cup \{\ell'_1, \dots, \ell'_s, \ell'\}\rangle_{\mathbb{F}_2}.\end{aligned}$$

Otherwise, C is a CNF clause, and there is an XNF clause $C_F \in F$ with $C = \{y_{i_\ell} \mid \ell \in C_F\}$ and $C|_{G_{\mathsf{unit}}} \equiv C_F$. Consider the reason clause decomposition $V_C = \big(V_C \cap \langle A \cup \{\ell_1, \ldots, \ell_{k-1}\}\rangle_{\mathbb{F}_2}\big) \oplus \langle \ell_C + 1\rangle_{\mathbb{F}_2}$ with a reason lineral ℓ_C. Then there is $C' \equiv C$ with $C' = \{b_1, \ldots, b_t\} \cup \{\ell_C\}$ and for each $j \in \{1, \ldots, t\}$ we have $b_j + 1 \in \langle A \cup \{\ell_1, \ldots, \ell_{k-1}\}\rangle_{\mathbb{F}_2}$ and $b_j \in \langle y_{i_\ell} + 1 \mid \ell \in C_F\rangle_{\mathbb{F}_2}$. Now, consider $C'_F = C'|_{G_{\mathsf{unit}}} \subseteq \mathbb{L}_n$. Then $C'_F \equiv C|_{G_{\mathsf{unit}}} = C_F$ and hence $C'_F \in F'$ by construction of F'. For $j \in \{1, \ldots, t\}$ let $c_j = b_j|_{G_{\mathsf{unit}}} \in C'_F$. Because of $y_{i_{c_j}} + c_j \in G_{\mathsf{unit}}$ we get

$$\begin{aligned} y_{i_{c_j}} + 1 \;&\in\; (c_j + 1) + \langle G_{\mathsf{unit}}\rangle_{\mathbb{F}_2} \;=\; (b_j + 1) + \langle G_{\mathsf{unit}}\rangle_{\mathbb{F}_2} \\ &\subseteq\; \langle G_{\mathsf{unit}}\rangle_{\mathbb{F}_2} + \langle A \cup \{\ell_1, \ldots, \ell_{k-1}\}\rangle_{\mathbb{F}_2}. \end{aligned}$$

Extend the lite lineral trail $[\ell'_1, \ldots, \ell'_s]$ with all elements in G_{unit} which are not yet included. Applying Lemma B.3 to this lite lineral trail shows that $y_{i_{c_j}} + 1 \in \mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}(G, A)$ for all $j \in \{1, \ldots, t\}$. Therefore, there is a lite lineral trail $[\ell'_1, \ldots, \ell'_r]$ for G under A with

$$G_{\mathsf{unit}} \cup \big\{y_{i_{c_j}} + 1 \mid j \in \{1, \ldots, t\}\big\} \;\subseteq\; A \cup \{\ell'_1, \ldots, \ell'_r\}.$$

By construction, $\{y_{i_{c_j}} \mid c_j \in C'_F\}$ is a CNF clause of G. It is a reason clause for $y_{i_{\ell_C}}$ under $A \cup \{\ell'_1, \ldots, \ell'_r\}$, and thus $[\ell'_1, \ldots, \ell'_r, y_{i_{\ell_C}}]$ is a lite lineral trail for G under A. Finally, with $y_{i_{\ell_C}} + \ell_C \in G_{\mathsf{unit}}$ we get

$$\begin{aligned} \ell_k \;=\; \ell_C|_{A \cup \{\ell_1, \ldots, \ell_{k-1}\}} \;&\in\; \ell_C + \langle A \cup \{\ell_1, \ldots, \ell_{k-1}\}\rangle_{\mathbb{F}_2} \\ &\subseteq\; y_{i_{\ell_C}} + \langle A \cup \{\ell'_1, \ldots, \ell'_r\}\rangle_{\mathbb{F}_2} \\ &\subseteq\; \langle A \cup \{\ell'_1, \ldots, \ell'_r, y_{i_{\ell_C}}\}\rangle_{\mathbb{F}_2}. \end{aligned}$$

This concludes the inductive proof, and the inclusion follows from Lemma B.3. □

Remark B.5. Proposition B.4 shows that using an extended CNF-XOR encoding with $\mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}$ can enhance its propagation power. In fact, it can find all literals implied by the set of linerals deduced by GCP. Thus, $\mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}$ can *simulate* GCP with respect to these implied literals. One should, however, keep in mind that this improvement comes at the price of an increased cost of propagation, as the extended CNF-XOR encoding of a k-XNF formula F contains up to $O(2^k)$ clauses and variables.

Remark B.6. By Remark 5.5, we know that for CNF-XOR formulae and variable assignments, algorithm $\mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}$ coincides with $\mathrm{BCP}^{\mathsf{XOR}}_{\mathsf{GJE}}$. Hence, Proposition B.4 shows that also $\mathrm{CDCL}^{\mathsf{XOR}}_{\mathsf{GJE}}$ can simulate GCP-like propagation of implied literals using an extended CNF-XOR encoding. Note, however, that this encoding generally come with an exponential increase in the number of clauses, see Remark B.2.

To see the proposition in action, consider the following example.

Example B.7. For the CNF-XOR formula $F = \big\{\{x_1 + x_2\}, \{x_1, x_2\}\big\} \subseteq \mathcal{P}(\mathbb{L}_2)$ we clearly have $\mathrm{GCP}^{\mathsf{lite}}{}_{\mathsf{Lex}}(F, \emptyset) = \{x_1 + x_2\}$ and $\mathrm{GCP}_{\mathsf{Lex}}(F, \emptyset) = \{x_1 + x_2, x_2\}$. This yields $\mathrm{ImpLits}\big(\mathrm{GCP}_{\mathsf{Lex}}(F, \emptyset)\big) = \{x_1, x_2\}$.

Let us now compute the trivial and extended CNF-XOR encodings.

(a) For the trivial encoding we use $\#\bigcup_{C \in F} C = \#\{x_1, x_2, x_1 + x_2\} = 3$ auxiliary variables. Then, we construct the formula

$$G \;=\; \big\{\{y_1 + x_1\}, \{y_2 + x_2\}, \{y_3 + x_1 + x_2\}, \{y_3\}, \{y_1, y_2\}\big\} \;\subseteq\; \mathcal{P}(\mathbb{L}_{2,3}).$$

It is easy to verify that $\mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}(G, \emptyset) \cap \mathbb{X}_2 = \emptyset$.

(b) To construct the extended encoding, we compute the reduced clauses in the equivalence classes of the XNF clauses of F. The one of $\{x_1 + x_2\}$ contains no other elements; for $\{x_1, x_2\}$ it contains the following 3 reduced clauses, see Remark 2.17.a.

$$\{x_1, x_2\} \;\equiv\; \{x_1 + x_2 + 1, x_2\} \;\equiv\; \{x_1 + x_2 + 1, x_1\}.$$

Collect the clauses in the XNF formula F' and compute its trivial CNF-XOR encoding. This requires $\#\bigcup_{C\in F'} C = \#\{x_1,\, x_2,\, x_1 + x_2,\, x_1 + x_2 + 1\} = 4$ auxiliary variables and results in a CNF-XOR formula $H \subseteq \mathcal{P}(\mathbb{L}_{2,4})$ with clauses

$$C_1 = \{y_1 + x_1\},\ \ C_2 = \{y_2 + x_2\},\ \ C_3 = \{y_3 + x_1 + x_2\},\ \ C_4 = \{y_4 + x_1 + x_2 + 1\}$$
$$C_5 = \{y_3\},\qquad C_6 = \{y_1, y_2\},\qquad C_7 = \{y_4,\, y_1\},\qquad C_8 = \{y_4,\, y_2\}.$$

Now, H is the extended CNF-XOR encoding of F.

An execution of $\mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}(H, \emptyset)$ may then produce the following lite lineral trail

$$[y_3^{C_5},\ y_3 + x_1 + x_2^{C_3},\ y_4 + x_1 + x_2 + 1^{C_4},\ y_4 + 1,\ y_1^{C_7},\ x_1^{C_1},\ x_2^{C_4}],$$

where the literal $y_4 + 1$ is deduced due to the extension of $\mathrm{GCP}^{\mathsf{lite}}$ with *Gauß-Jordan* reasoning as the sum of all prior linerals. This shows

$$\mathrm{GCP}^{\mathsf{lite}}_{\mathsf{GJE}}(H, \emptyset) \cap \mathbb{X}_2 \;=\; \{x_1, x_2\} \;=\; \mathrm{ImpLits}\big(\mathrm{GCP}_{\mathsf{Lex}}(F, \emptyset)\big).$$

Received 02 September 2025; accepted 26 June 2026