On Conversions from CNF to ANF

Jan Horáček and Martin Kreuzer

Faculty of Informatics and Mathematics
University of Passau, D-94030 Passau, Germany
Jan.Horacek@uni-passau.de, Martin.Kreuzer@uni-passau.de

Abstract. In this paper we discuss conversion methods from the conjunctive normal form (CNF) to the algebraic normal form (ANF) of a Boolean function. Whereas the reverse conversion has been studied before, the CNF to ANF conversion has been achieved predominantly via a standard method which tends to produce many polynomials of high degree. Based on a block-building mechanism, we design a new blockwise algorithm for the CNF to ANF conversion which is geared towards producing fewer and lower degree polynomials. In particular, we look for as many linear polynomials as possible in the converted system and check that our algorithm finds them. Experiments show that the ANF produced by our algorithm outperforms the standard conversion in "real life" examples originating from cryptographic attacks.

Keywords: conjunctive normal form, algebraic normal form, Boolean polynomial, Boolean Gröbner basis, SAT solving

1 Introduction

The Boolean satisfiability problem (SAT) and polynomial system solving over a finite field (PSS) are two fundamental problems of propositional logic and computational commutative algebra, respectively. The *decision* versions of these problems, i.e., whether there exists a satisfying assignment for a Boolean formula or a common zero for a polynomial system over a finite field, are known to be NP-complete. In other words, both problems are in NP and reducible to each other in polynomial time. For practical reasons, the *search* versions of these problems are very important, i.e., to find one or all satisfying assignments, or to find one or all common zeros.

Efficient conversion methods for transforming a Boolean polynomial system to a SAT instance have been studied carefully. Some of them are tailored to algebraic attacks in cryptography (see [3] and [16]). Implementations of these conversions can be found for instance in the computer algebra systems `Sage` (see [21]) and `ApCoCoA` (see [20]). In this paper we focus on the reverse direction, i.e., the transformation of a SAT instance given by a set of clauses of a formula in conjunctive normal form (CNF) to a set of Boolean polynomials in algebraic normal form (ANF). The standard way to perform this conversion clause by clause has been known for a long time (see for instance [15]). However,

more efficient methods which try to combine several clauses into one Boolean polynomial, preferably a short polynomial of low degree, have been considered only more recently and cursorily (see for instance [7]), mostly in the context of integrating Gröbner basis techniques into the various stages of a SAT solver.

Our motivation for studying more efficient methods for the CNF to ANF conversion derives from the attempt to solve large instances of the SAT or PSS problem which originate from cryptographic attacks. Rather than using some heuristics to call a Gröbner basis solver for supporting a SAT solver at selected points in the CDCL algorithm (for example, as in [22] and [10]), we are running a SAT solver and an algebraic solver in parallel and let them interchange information. For instance, for algebraic fault attacks targeting the Small-Scale AES cryptosystem, SAT clauses and Boolean polynomials have been derived both from a VDHL implementation of the circuit and from a functional model in [13]. It has been observed that a combination of the SAT solver `antom` (cf. [18]) and a border basis solver (cf. [14]) outperforms the individual solvers, if the communication between the processes provides clauses resp. Boolean polynomials which are most likely to aid the solver. Hence we are most interested in CNF to ANF conversions which yield many short linear or quadratic polynomials.

To achieve this goal we proceed as follows. In Section 2 we recall the basic definitions and notations regarding the ring of Boolean polynomials and propositional logic. In Section 3 we recall some conversion methods from ANF to CNF, and in Section 4 we briefly recap the standard conversion from CNF to ANF. Our main algorithm for the CNF to ANF conversion is developed in Section 5. The idea is to combine all clauses which share a certain number of variables (irrespective of their sign) into a block of clauses. Using an *overlapping number* m, we introduce an efficient algorithm for producing these m-blocks. Then, in the blockwise conversion algorithm, these m-blocks are converted individually using the standard conversion and for each converted block of polynomials we compute a reduced Gröbner basis. As it turns out, the Gröbner bases contain many more low degree polynomials than a standard conversion would have provided us with. We also show that the computation of these reduced Gröbner bases encompasses algebraic versions of the usual DPLL rules of inference (cf. Prop. 4).

The task of finding as many linear polynomials as possible in the ANF conversion of a block of clauses is examined further in Section 6. Based on the structure of the CNF transformation of a linear polynomial, we derive a combinatorial test which checks whether a block of clauses contains sufficiently many clauses that can be extended suitably to cover the transformation of a linear polynomial (cf. Prop. 7). This test is implemented in Algorithm 4. However, as Prop. 8 shows, these linear polynomials are found also by the calculation of the blockwise reduced Gröbner bases. Thus the blockwise conversion method given in Algorithm 3 automatically contains all linear polynomials which can be deduced from the set of clauses by simple combinatorial methods. An amusing consequence is that sometimes the double conversion from ANF to CNF and back is sufficient to solve the Boolean polynomial system, because the block-

wise conversion algorithm produces enough linear polynomials to allow Gaußian elimination to work.

The final section contains some experiments in which we compare the new blockwise conversion algorithm to the standard CNF to ANF conversion. In many examples originating from cryptographic attacks and factorization problems, the new conversion method improves the input for the algebraic solver substantially.

Unless stated otherwise, we use the basic definitions and notation of [17].

2 Background

In this section we recall basic definitions and known results and introduce useful notation. In the following we let $\mathbb{F}_2$ be the field of two elements and $\mathbb{F}_2[x_1, \ldots, x_n]$ a polynomial ring over $\mathbb{F}_2$. The ideal $F = \langle x_1^2+x_1, \ldots, x_n^2+x_n\rangle$ is called the **field ideal**, since it is the vanishing ideal of $\mathbb{F}_2^n$. The ring $\mathbb{B}_n = \mathbb{F}_2[x_1, \ldots, x_n]/\langle F\rangle$ is called the **ring of Boolean polynomials** in the indeterminates $x_1, \ldots, x_n$. We assume that its elements are represented by polynomials whose support consists only of squarefree terms. Boolean polynomials in this shape are said to be in **algebraic normal form (ANF)**. An arbitrary polynomial f can be transformed to ANF by computing its normal form $\mathrm{NF}_F(f)$ with respect to the field ideal. Notice that we use "$+$" instead of "$\oplus$" for addition in $\mathbb{F}_2$.

Given a set $S = \{f_1, \ldots, f_s\} \subseteq \mathbb{B}_n$, we define the **set of $\mathbb{F}_2$-rational zeros** of S by

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

In fact, the set $\mathcal{Z}(S)$ does not depend on the particular choice of generators of the ideal $I = \langle f_1, \ldots, f_s\rangle$, but only on the ideal itself. Thus we can write $\mathcal{Z}(I)$. Boolean polynomials correspond 1-1 to Boolean functions. The only Boolean polynomial f with $\mathcal{Z}(f) = \emptyset$ is the constant polynomial 1.

Solvers that allow us to describe the set of zeros of a given ideal by algebraic techniques are referred to as *algebraic solvers*. We mention here the Boolean Gröbner Basis Algorithm [5, Ch. 2], the Boolean Border Basis Algorithm [14], the XL/XSL algorithm and its variants [8], and ElimLin [9]. For the (Boolean) Gröbner Basis Algorithm, the library `PolyBoRi` [6] and the `FGb` library [12] provide efficient actual implementations of such solvers. The basic principle of algebraic solvers is to generate new polynomials in the ideal and simplify the newly derived polynomials by the old ones. For the theory of Boolean Gröbner bases, we refer to [5, Ch. 2]. Every ideal $I \subseteq \mathbb{B}_n$ can be represented by an ideal $I' \subseteq \mathbb{F}_2[x_1, \ldots, x_n]$ such that $F \subseteq I'$. Hence Boolean Gröbner bases correspond to Gröbner bases of ideals containing the field equations.

Every propositional logic formula φ can be encoded in **conjunctive normal form (CNF)**. A clause is a set of literals, i.e. logical variables X_i or their negation $\bar{X}_i$. A set of clauses $C = \big\{\{L_{1,1}, \ldots, L_{1,n_1}\}, \ldots, \{L_{k,1}, \ldots, L_{k,n_k}\}\big\}$ corresponds to the logic formula $\varphi = (L_{1,1} \vee \cdots \vee L_{1,n_1}) \wedge \cdots \wedge (L_{k,1} \vee \cdots \vee L_{k,n_k})$. We always assume that φ is in CNF and given by its set of clauses C. This allows us to identify φ and C.

Next we define the **set of satisfying assignments** for a given set of clauses C in n logical variables by

$$\mathrm{SAT}(C) = \big\{a \in \{\texttt{False}, \texttt{True}\}^n \mid C(a) \text{ evaluates to } \texttt{True}\big\}.$$

The algorithms that search for a satisfying assignment for C are called *SAT solvers*. In order to find the whole solution space, so-called #SAT solvers are used. Most modern SAT solvers are based on a CDCL procedure, i.e. on resolution with the addition of clause learning. They generate new clauses, called conflict clauses, that guide the computation. Together with non-chronological backtracking and highly optimized data structures, they are very powerful tools. Two standard implementations of the SAT algorithm are `MiniSAT` [11] and `Glucose` [1].

To distinguish variables in a Boolean polynomial ring and in formulae, we use lower-case letters for variables in Boolean polynomials and capital letters for the corresponding logical variables. Moreover, we identify $\texttt{True} \equiv 1$ and $\texttt{False} \equiv 0$.

Definition 1. *Let $S \subseteq \mathbb{B}_n$ be a set of Boolean polynomials and C a set of clauses in the logical variables $X_1, \ldots, X_n$. We say that C is a* **logical representation** *of S, resp. S is an* **algebraic representation** *of C, if and only if* $\mathrm{SAT}(C) = \mathcal{Z}(S)$.

The algebraic representation of S is not unique in general. On the other hand, if $\#S = 1$, the representation is unique. Namely, one Boolean polynomial f represents the unique Boolean function $\mathbb{F}_2^n \to \mathbb{F}_2$ mapping $a \mapsto 0$ if $a \in \mathrm{SAT}(C)$ and $a \mapsto 1$ otherwise. In this case, we say that f is the **standard algebraic representation** of C.

3 Conversions from ANF to CNF

First of all, let us discuss what kinds of ANF systems are well suited to converting them to SAT. Note that if the system is rather dense, it is probably better to solve it by algebraic solvers or even by brute force. A typical example where algebraic solvers outperform SAT solvers is solving a dense linear system. On the other hand, the memory consumption of SAT solvers is kept under control, and therefore they tend to be faster for sparse constraint inputs, for which algebraic solvers may have a huge space consumption. (For more details and experiments, see [2], Ch. 13.)

In this section we recall some efficient conversion methods for a set of Boolean polynomials in ANF (i.e., XOR of ANDs) to a Boolean formula in CNF (i.e., AND of ORs). There are basically two types of such conversions. Both of them convert only one Boolean polynomial at a time. The first conversion method does not introduce new auxiliary variables, creating a sparse representation, whereas the second one does and results in a dense representation.

The **sparse conversion** is truth-based and uses the assumption that the input polynomials are rather sparse (i.e. they do not have many terms and variables). Thus one can go trough all possible assignments for all logical variables contained in the polynomial to construct the sparse CNF.

Example 1. Consider the truth table of the polynomial $f(x_1, x_2) = x_1x_2+x_2+1$.

x_1	x_2	f
0	0	1
0	1	0
1	0	1
1	1	1

For each assignment that yields `True`, we construct one clause that eliminates this particular assignment. Thus the set of clauses $C = \{\{X_1, X_2\}, \{\bar{X}_1, X_2\}, \{\bar{X}_1, \bar{X}_2\}\}$ is the logical representation of the polynomial f. Note that the set $\{x_1, x_2 + 1\}$ is an algebraic representation of C as well, so the representations are not uniquely determined.

Dense conversion methods (see [3], [16]) introduce new variables. Foremost, the polynomial $f \in \mathbb{B}_n$ is linearized. For each of its terms of degree greater than one, we introduce a new auxiliary indeterminate t and encode the resulting binomial in CNF. E.g., we convert $x_1x_2x_3$ by encoding $t + x_1x_2x_3$ to the clauses $\{X_1, \bar{T}\}$, $\{X_2, \bar{T}\}$, $\{X_3, \bar{T}\}$, $\{\bar{X}_1, \bar{X}_2, \bar{X}_3, T\}$. After this step, we are left with (possibly long) linear polynomials. We split them into smaller ones by introducing further auxiliary indeterminates according to a predefined cutting number r. To the resulting shorter linear polynomials we apply the sparse conversion.

Example 2. Let $r = 3$. We cut the linear polynomial $x_1 + x_2 + \cdots + x_5$ into two polynomials $x_1 + x_2 + x_3 + y$ and $y + x_4 + x_5$. Note that we have introduced one new indeterminate y here. For instance, when we convert $x_1 + x_2 + x_3 + x_4$, we get the clauses

$$\{\bar{X}_1, X_2, X_3, X_4\}, \{X_1, \bar{X}_2, X_3, X_4\}, \{X_1, X_2, \bar{X}_3, X_4\}, \{X_1, X_2, X_3, \bar{X}_4\},$$
$$\{\bar{X}_1, \bar{X}_2, \bar{X}_3, X_4\}, \{\bar{X}_1, \bar{X}_2, X_3, \bar{X}_4\}, \{\bar{X}_1, X_2, \bar{X}_3, \bar{X}_4\}, \{X_1, \bar{X}_2, \bar{X}_3, \bar{X}_4\}.$$

Both conversions suffer from the problem that breaking the XOR structure in the ANF tends to introduce many auxiliary indeterminates or many new clauses.

4 The Standard Conversion from CNF to ANF

The standard conversion from CNF to ANF converts each clause of C to one Boolean polynomial. It has been known for a long time (cf. [15]). The detailed description is given in Algorithm 1.

Proposition 1. *Algorithm 1 outputs a system of Boolean polynomials S such that S is an algebraic representation of C.*

Proof. Let $c = \{L_1, L_2, \ldots, L_m\}$ be a clause of C. The assignment $(a_1, \ldots, a_n) \in \mathbb{F}_2^n$ satisfies c if and only if the polynomial $f = \ell_1 \cdots \ell_m$ vanishes at the point $(a_1, \ldots, a_n)$, where $\ell_i = x_i + 1$ for $L_i = X_i$ and $\ell_i = x_i$ for $L_i = \bar{X}_i$. □

Let us apply this algorithm to a concrete case.

Algorithm 1 (Standard CNF to ANF Conversion)
Input: A set C in logical variables $X_1, \ldots, X_n$.
Output: A set $S \subseteq \mathbb{B}_n$ such that S is an algebraic representation of C.

```
 1: S := ∅
 2: foreach c in C do
 3:    f := 1
 4:    foreach L in c do
 5:       if L = X_i is positive then
 6:          f := f · (x_i + 1)
 7:       else if L = X̄_i is negative then
 8:          f := f · (x_i)
 9:       end if
10:    end foreach
11:    S := S ∪ {f}
12: end foreach
13: return S
```

Example 3. Given the set of clauses $\{\{X_1, X_2\}, \{\bar{X}_1, X_2, X_3\}, \{X_4, X_5\}, \{X_1, \bar{X}_2, X_3\}, \{\bar{X}_1, \bar{X}_2, \bar{X}_3\}, \{X_4, \bar{X}_5\}$, the Standard CNF to ANF Conversion yields the following results:

$$\begin{array}{lcl} \{X_1, X_2\} & \to & x_1x_2 + x_1 + x_2 + 1 \\ \{\bar{X}_1, X_2, X_3\} & \to & x_1x_2x_3 + x_1x_2 + x_1x_3 + x_1 \\ \{X_4, X_5\} & \to & x_4x_5 + x_4 + x_5 + 1 \\ \{X_1, \bar{X}_2, X_3\} & \to & x_1x_2x_3 + x_1x_2 + x_2x_3 + x_1 \\ \{\bar{X}_1, \bar{X}_2, \bar{X}_3\} & \to & x_1x_2x_3 \\ \{X_4, \bar{X}_5\} & \to & x_4x_5 + x_5 \end{array}$$

Clearly, Algorithm 1 performs at most $n \cdot \#C$ multiplications in $\mathbb{B}_n$. Thus it is of polynomial time complexity. Notice that its output for a single input clause c is the standard algebraic representation of c. Moreover, $\deg(f)$ equals the length of the clause c in Step 12 of the algorithm. Hence even a small set of clauses may be converted to a polynomial system containing high-degree polynomials. The degree and the length of the support of these polynomials can be viewed as an indicator of their usefulness. It follows that such a conversion does, in general, not give an encoding which is useful for further applications. In particular, converting a simple Boolean system S to CNF using the sparse strategy and back to polynomials by Algorithm 1, gives us a rather denser and higher-degree system $\hat{S}$. Even if the computation of a Gröbner basis of the ideal $\langle S \rangle$ may be done in seconds, a Gröbner basis of the ideal $\langle \hat{S} \rangle$ can take hours to compute.

5 A Blockwise Conversion from CNF to ANF

Let C be a set of clauses representing a propositional logic formula in CNF. First of all, we group certain clauses in C together using the following definitions.

Definition 2. *(a) The set of variables X_i such that X_i or $\bar{X}_i$ is contained in one of the clauses of C is denoted by $\mathrm{var}(C)$ and is called the* **set of variables** *of C.*
(b) We say $c \in C$ has **positive** *(resp.* **negative** *) sign if the number of negative literals is an even (resp. odd) number.*
(c) We define the **length of a clause** *$c \in C$ as the cardinality $\#c$.*
(d) Let $c, c' \in C$. A number $m \geq 1$ such that $\#\big(\mathrm{var}(c) \cap \mathrm{var}(c')\big) \geq m$ is called an **overlapping number** *of c and c'.*

Given a number m, Algorithm 2 decomposes a set of clauses C into blocks B_c for $c \in C$ such that m is an overlapping number of c with every clause in B_c.

Algorithm 2 (Building m-Blocks)
Input: A set of clauses C, an overlapping number $m \in \mathbb{N}$.
Output: A set of subsets $\mathcal{B}$ of C and a subset T of C such that for $B \in \mathcal{B}$ with $\#B \geq 2$ and for every $b \in B$, there exists an element $b' \in B \setminus \{b\}$ with the property that m is an overlapping number for b and b', and such that $\big(\bigcup_{B \in \mathcal{B}} B\big) \cup T = C$ and every clause in T contains less than m literals.

1: **foreach** c **in** C **do**
2: $\quad B_c := \big\{c' \in C \mid \#\big(\mathrm{var}(c) \cap \mathrm{var}(c')\big) \geq m\big\}$
3: **end foreach**
4: $\mathcal{B}' := \{B_c \mid c \in C,\ B_c \neq \emptyset\}$
5: Let $\mathcal{B}$ be the set of maximal elements of $\mathcal{B}'$ w.r.t. inclusion.
6: $T := C \setminus \bigcup_{c \in C} B_c$
7: **return** $(\mathcal{B}, T)$

Notice that some clauses in C may not be included in the set $\mathcal{B}$ produced by Algorithm 2. This happens when the length of a clause is less than m. Such clauses are returned in the set T. Furthermore, the cardinality of the set of clauses contained in $\mathcal{B} \cup T$ may be greater than $\#C$. The cardinality is at least equal to $\#C$, because every $c \in C$ is contained either in the set B_c in $\mathcal{B}$, or c is put into T in Step 6. Moreover, we note that Algorithm 2 performs at most $\#C$ iterations of the foreach loop, at most $\binom{\#C}{2}$ intersections in Step 2, and at most $\binom{\#\mathcal{B}}{2}$ comparisons in Step 5. Hence this algorithm has a polynomial time complexity.

Proposition 2. *The output of Algorithm 2 is uniquely determined.*

Proof. The sets in $\mathcal{B}$ are related to the following graph. For $m \in \mathbb{N}$, we define an undirected graph $\mathcal{G}_{m,C}$ which has C as vertices and for which two distinct clauses $c, c' \in C$ form an edge if and only if $\#\big(\mathrm{var}(c) \cap \mathrm{var}(c')\big) \geq m$. Clearly, Step 2 of Algorithm 2 computes the closed neighborhood of a vertex c of $\mathcal{G}_{m,C}$, i.e. the set of all vertices connected to c by an edge. Then Step 5 selects the maximal neighborhoods w.r.t. inclusion. This shows that the output of Algorithm 2 is uniquely determined by C and m, and does not depend on the order in which the clauses c are selected in Step 1. □

The elements of the set $\mathcal{B}$ returned by Algorithm 2 will be called the m-**blocks** of C. Let us apply Algorithm 2 in some easy cases.

Example 4. Let $C = \{c_1, c_2, c_3\}$ with $c_1 = \{X_1, X_2, X_3, X_4\}$, $c_2 = \{X_1, X_2\}$ and $c_3 = \{X_3, X_4\}$, and let $m = 2$. Then the entire set C is one 2-block. Notice that this block does not correspond to a complete subgraph of $\mathcal{G}_{m,C}$, because the edge (c_2, c_3) is missing.

Example 5. In the setting of Example 3, Algorithm 2 calculates the following two 2-blocks.

$$\begin{matrix}\{X_1, X_2\}\\ \{\bar{X}_1, X_2, X_3\}\\ \{X_4, X_5\}\\ \{X_1, \bar{X}_2, X_3\}\\ \{\bar{X}_1, \bar{X}_2, \bar{X}_3\}\\ \{X_4, \bar{X}_5\}\end{matrix} \quad \rightarrow \quad \begin{bmatrix}\{X_1, X_2\}\\ \{\bar{X}_1, X_2, X_3\}\\ \{X_1, \bar{X}_2, X_3\}\\ \{\bar{X}_1, \bar{X}_2, \bar{X}_3\}\end{bmatrix}, \begin{bmatrix}\{X_4, X_5\}\\ \{X_4, \bar{X}_5\}\end{bmatrix}$$

Let σ be a degree compatible term ordering. We say that a set of Boolean polynomials $G \subseteq \mathbb{B}_n$ is LT_σ**-interreduced** if $\mathrm{LT}_\sigma(g) \neq \mathrm{LT}_\sigma(g')$ for all $g, g' \in G$ with $g \neq g'$. Given an arbitrary set of Boolean polynomials $G \subseteq \mathbb{B}_n$, we can LT_σ-interreduce G via Gaußian elimination on the coefficient matrix of G, where its columns are sorted from biggest to the smallest term w.r.t. σ. Now we are ready to describe the main Algorithm 3.

Algorithm 3 (Blockwise CNF to ANF Conversion)
Input: A set of clauses C in logical variables $X_1, \ldots, X_n$, a degree compatible term ordering σ, and an overlapping number $m \in \mathbb{N}$.
Output: A set $S_{\sigma,m} \subseteq \mathbb{B}_n$ such that $S_{\sigma,m}$ is an algebraic representation of C.
Requires: Algorithm 1 and 2, a reduced Boolean Gröbner basis algorithm.

1: $S' := \emptyset$
2: Using Algorithm2(C, m), compute a pair $(\mathcal{B}, T)$.
3: $\mathcal{B} := \mathcal{B} \cup \bigcup_{t \in T} \{t\}$
4: **foreach** B **in** $\mathcal{B}$ **do**
5: $\quad Q :=$ Algorithm1(B)
6: $\quad$ Let G be the reduced Boolean σ-Gröbner basis of the ideal $\langle Q \rangle$, i.e., the reduced Boolean Gröbner basis with respect to the term ordering σ.
7: $\quad S' := S' \cup G$
8: **end foreach**
9: Let $S_{\sigma,m}$ be an LT_σ-interreduced $\mathbb{F}_2$-basis of $\langle S' \rangle_{\mathbb{F}_2}$ such that its coefficient matrix w.r.t. σ is in reduced row echelon form.
10: **return** $S_{\sigma,m}$

It is difficult to give a meaningful upper bound for the time complexity of this algorithm, since it involves a number of Gröbner basis calculations. As we shall see in the next section, if one of the sets in $\mathcal{B}$ contains a complete signed

set of clauses (see Definition 3), the conversion will contain a linear polynomial. In this case, the corresponding Gröbner basis will be found rather quickly. As one can infer from the tables in the last section, this happens a lot in practically relevant cases. But, of course, it is clear that one can construct special sets of clauses for which the Gröbner basis calculation is particularly expensive.

Proposition 3. *The output of Algorithm 3 is an algebraic representation of C and is uniquely determined by σ and m.*

Proof. First we prove that $S_{\sigma,m}$ is an algebraic representation of C. In Step 3 we have $\bigcup_{B\in\mathcal{B}} \left(\bigcup_{c\in B} c\right) = C$, because every $c \in C$ is contained either in the set B_c in $\mathcal{B}$, or c is put into T in Step 6 of Algorithm 2. We know that Q is an algebraic representation of $B \in \mathcal{B}$ in Step 5 by Proposition 1. Furthermore, G is an algebraic representation of B as well, because $\langle Q\rangle = \langle G\rangle$. Clearly, if G_1 resp. G_2 are algebraic representations of B_1 resp. B_2, then $G_1 \cup G_2$ is an algebraic representation of $B_1 \cup B_2$. Thus S' is an algebraic representation of C in Step 9. LT_σ-interreduction does not change the set of zeros, and therefore $S_{\sigma,m}$ is an algebraic representation of C.

By Proposition 2 we know that the set $\mathcal{B}$ in Step 3 is uniquely determined. The reduced Boolean σ-Gröbner basis of the ideal $\langle Q\rangle$ in Step 6 is unique, and so is the basis in Step 9. □

Example 6. Let us apply Algorithm 3 in the setting of Example 3.

$$\begin{bmatrix} \{X_1, X_2\} & \to & x_1x_2 + x_1 + x_2 + 1 \\ \{\bar{X}_1, X_2, X_3\} & \to & x_1x_2x_3 + x_1x_2 + x_1x_3 + x_1 \\ \{X_1, \bar{X}_2, X_3\} & \to & x_1x_2x_3 + x_1x_2 + x_2x_3 + x_1 \\ \{\bar{X}_1, \bar{X}_2, \bar{X}_3\} & \to & x_1x_2x_3 \end{bmatrix} \quad \to \quad \begin{matrix} x_2x_3 + x_2 + x_3 + 1 \\ x_1 + x_2 + x_3 \end{matrix}$$

$$\begin{bmatrix} \{X_4, X_5\} & \to & x_4x_5 + x_4 + x_5 + 1 \\ \{X_4, \bar{X}_5\} & \to & x_4x_5 + x_5 \end{bmatrix} \quad \to \quad x_4 + 1$$

As we can see, the output is a set of three polynomials of degrees 1,1,2 instead of the six polynomials of degrees 2,2,2,3,3,3 in Example 3.

In the following proposition we study Step 6 of Algorithm 3 in more detail. Its Claims (1)-(4) are algebraic versions of the one-literal, subsumption, clean-up, and resolution rules of DPLL. In this sense, the Gröbner basis algorithm can be interpreted as performing simple logical reasoning.

Proposition 4. *In the setting of Algorithm 3, let $B = \{c_1, \ldots, c_k\}$ be a set of clauses. Let $Q = \{q_1, \ldots, q_k\}$ be the set of Boolean polynomials such that q_i is the standard algebraic representation of c_i for $i = 1, \ldots, k$. Let G be the reduced Boolean σ-Gröbner basis of the ideal $I = \langle Q\rangle$.*

(1) Let $c_i, c_j \in B$ be clauses such that c_i is a proper subclause of c_j. Then G is equal to the reduced Boolean σ-Gröbner basis of $\langle Q \setminus \{q_j\}\rangle$.

(2) *Let* L *be a literal and assume that* $c_j = \{L\}$ *is an element of* B. *Let* $\{q_{i_1}, \ldots, q_{i_s}\}$ *be the set of all clauses in* B *different from* q_j *and containing the literal* L. *Then* G *is the reduced Boolean* σ*-Gröbner basis of* $\langle Q \setminus \{q_{i_1}, \ldots, q_{i_s}\}\rangle$.

(3) *Let* $c_i \in B$ *be of the form* $c_i = c' \cup \{X_j, \bar{X}_j\}$ *for some clause* c' *and a logical variable* X_j. *Then* G *is the reduced Boolean* σ*-Gröbner basis of* $\langle Q \setminus \{q_i\}\rangle$.

(4) *Assume that* $c_i, c_j \in B$ *satisfy* $c_i = w \cup \{X_e\}$ *and* $c_j = w' \cup \{\bar{X}_e\}$ *for some logical variable* X_e *and clauses* w, w'. *Let* $r = w \cup w'$ *be the resolvent of* c *and* c' *on the variable* X_e. *Then the standard algebraic representation of* r *is the S-polynomial of* q_i, q_j.

(5) *We have* $\mathrm{SAT}(B) = \emptyset$ *if only if* $G = \{1\}$.

(6) *Let* $\#B \geq 2$. *If there exists a clause* $c_j \in B$ *such that* $\mathrm{var}(c') \subseteq \mathrm{var}(c_j)$ *holds for all* $c' \in B$, *then we have* $\max\{\deg(g) \mid g \in G\} < \max\{\deg(q) \mid q \in Q\}$.

(7) *Let* $f \in I$ *be such that there exists a clause* c *for which* f *is the standard algebraic representation of* c. *If there exists a Boolean polynomial* $g \in G$ *such that* $\mathrm{LT}_\sigma(f) = \mathrm{LT}_\sigma(g)$, *then* $f = g$.

Proof. (1) From the inclusion $c_i \subset c_j$, we know that q_j is a multiple of q_i. Hence q_j is reduced to zero by q_i and the claim follows.

(2) All polynomials in $\{q_{i_1}, \ldots, q_{i_s}\}$ are multiples of q_j and thus are reduced to zero.

(3) The standard algebraic representation of c_i is $q_i = x_j(x_j + 1)f$ for some variable x_j and a polynomial f. Thus the Boolean polynomial q_i satisfies $q_i = (x_j^2 + x_j)f = 0$ in $\mathbb{B}_n$ and the claim follows.

(4) The standard algebraic representation of c_i resp. c_j is $q_i = x_e f$ resp. $q_j = (x_e + 1)g$ for some polynomials f, g. The S-polynomial of q_i, q_j is equal to $fg = g(x_e f) + f(x_e + 1)g$.

(5) We know that the variety of Q over the algebraic closure of $\mathbb{F}_2$ is equal to $\mathcal{Z}(Q)$, because we assume that the field equations are included in the ideal. Hence the claim follows from the strong version of Hilbert's Nullstellensatz.

(6) If q_j is the only polynomial with the maximal degree in Q, then there exists $c_i \in B$ such that $c_i \subset c_j$ and we apply Claim (1). If there is another $q \in Q$ with the maximal degree in Q different from q_j, then $\mathrm{LT}_\sigma(q) = \mathrm{LT}_\sigma(q_j)$. Hence we drop the degree at least by one after the LT-reduction.

(7) We have $f = \prod_{j=1}^{\ell}(x_{i_j} + a_j)$ for some $a_j \in \mathbb{F}_2$ and for some number ℓ. Because $\mathrm{LT}_\sigma(f) = \mathrm{LT}_\sigma(g)$, we know that $\mathrm{LT}_\sigma(f)$ is minimal w.r.t. division in $\mathrm{LT}_\sigma(I)$. All terms in f divide $\mathrm{LT}_\sigma(f)$, and so f can not be further reduced. Thus f is contained in some reduced Boolean σ-Gröbner basis of the ideal I. The claim follows form the uniqueness of the reduced Boolean σ-Gröbner basis. □

Notice that Claim (6) can also be found in [5, Thm. 5.3.5.].

6 Conversion to Linear Polynomials

The most valuable polynomials for algebraic solvers in the result of a CNF to ANF conversion algorithm are the linear ones. Therefore we now focus on the

problem of identifying sets of clauses containing a linear polynomial in their algebraic representation.

Definition 3. *A set of clauses, all of which have the same length, which consists of all possible clauses with either only positive or only negative sign is called a* **complete signed set** *of clauses.*

A complete signed set of clauses forms a complete subgraph of the graph $\mathcal{G}_{\ell,C}$ (see the definition in the proof of Proposition 2) having only positive, or only negative clauses of length ℓ as nodes. A complete signed set of clauses of length ℓ consists of $2^{\ell-1}$ clauses.

Proposition 5. *Let K be a complete signed set of clauses with positive (resp. negative) sign and $\operatorname{var}(K) = \{X_{i_1}, \ldots, X_{i_\ell}\}$. Then $x_{i_1} + \cdots + x_{i_\ell} + 1$ (resp. $x_{i_1} + \cdots + x_{i_\ell}$) is the standard algebraic representation of K.*

Proof. Let K' be the sparse conversion of $f = x_{i_1} + \cdots + x_{i_\ell} + 1$. From the truth table of f it is easy to see that K' is a complete signed set of clauses with positive sign in the variables $\operatorname{var}(K)$. Complete signed sets of clauses with positive sign are uniquely determined by their set of variables. Thus we get $K = K'$. The negative case follows analogously. □

Example 2 illustrates the previous proposition. A lower number of clauses can also produce linear polynomials, but we have to allow clauses of different lengths.

Proposition 6. *(a) Let φ, ψ be propositional logic formulas. Then we have $\varphi \equiv (\varphi \vee \psi) \wedge (\varphi \vee \bar{\psi})$.*
(b) Let c, w be clauses. The set $\{c\}$ is equivalent to $\{c \cup w, c \cup \bar{w}\}$.
(c) In the setting of (b), assume that w has length k. Write $w = \{L_1, \ldots, L_k\}$ with literals L_i. Then the set $\{c\}$ is equivalent to the set of all 2^k clauses of shape $c \cup \{L_1^*\} \cup \cdots \cup \{L_k^*\}$, where L_i^* equals to L_i or $\bar{L}_i$.

Proof. Claim (a) can be easily proven by comparing a truth table for φ and $(\varphi \vee \psi) \wedge (\varphi \vee \bar{\psi})$. The other claims are immediate consequences of (a). □

The following example illustrates this proposition.

Example 7. Let $B = \big\{\{X_1, X_2\}, \{\bar{X}_1, X_2, X_3\}, \{X_1, \bar{X}_2, X_3\}, \{\bar{X}_1, \bar{X}_2, \bar{X}_3\}\big\}$. The first clause in B is equivalent to the two clauses $\{X_1, X_2, X_3\}$, $\{X_1, X_2, \bar{X}_3\}$. In view of this, we have covered all four possible combinations for negative signed clauses of length 3. Indeed, Algorithm 3 converts B into $x_1 + x_2 + x_3$ and $x_2x_3 + x_2 + x_3 + 1$.

Proposition 6 leads to the following combinatorial test for checking whether a set of clauses B converts to a linear polynomial.

Proposition 7. *Let $B = \{c_1, \ldots, c_k\}$ be a set of clauses and $V = \{X_{i_1}, \ldots, X_{i_\ell}\}$ be a set of ℓ variables in $\mathrm{var}(B)$. For $j = 1, \ldots, k$ define the following sets:*

$$B^+_{j,V} = \{c \subseteq V \cup \bar{V} \mid c_j \subseteq c, \#c = \ell, c \text{ has positive sign}\},$$
$$B^-_{j,V} = \{c \subseteq V \cup \bar{V} \mid c_j \subseteq c, \#c = \ell, c \text{ has negative sign}\}.$$

(1) If

$$2^{\ell-1} = \sum_{\emptyset \neq J \subseteq \{1,2,\ldots,k\}} (-1)^{\#J-1} \cdot \#\Big(\bigcap_{j \in J} B^+_{j,V}\Big),$$

then the ideal generated by any algebraic representation of B contains $x_{i_1} + \cdots + x_{i_\ell} + 1$.

(2) If

$$2^{\ell-1} = \sum_{\emptyset \neq J \subseteq \{1,2,\ldots,k\}} (-1)^{\#J-1} \cdot \#\Big(\bigcap_{j \in J} B^-_{j,V}\Big),$$

then the ideal generated by any algebraic representation of B contains $x_{i_1} + \cdots + x_{i_\ell}$.

Proof. Let us focus on the first (i.e., positive) case. The second case is analogous. In view of Proposition 6, the set $B^+_{j,V}$ contains all possible extensions of c_j in variables V to positive clauses of length ℓ. We search for a complete signed set in the union of all sets $B^+_{j,V}$. In other words, the cardinality of this union must be equal to $2^{\ell-1}$ in order to contain a complete signed set. Note that these sets may not be disjoint. Thus we use the inclusion-exclusion principle for determining $\#\big(\bigcup_{j=1}^k B^+_{j,V}\big)$. □

In practice we do not have to apply the inclusion-exclusion principle, if the programming language we use has "set" as a built-in data structure. Algorithm 4 is a straight-forward application of Proposition 7. The sets $B^+_{j,V}$ and $B^-_{j,V}$ can be computed by extensions to the prescribed length ℓ via brute-force and grouping the result according to sign. Note that the ideas behind Algorithm 4 can be further developed, and a more efficient algorithm can be designed. (Some attempts in this direction can be deduced from the source code of `CryptoMiniSat`, see `src/xorfinder.cpp` in [19].)

Since we have to check all subsets of $\mathrm{var}(B)$ in Step 2, Algorithm 4 is only practical for rather small-sized sets $\mathrm{var}(B)$. Even if we are still able to directly derive linear polynomials from a large set of clauses C by Algorithm 4, the following proposition shows that Algorithm 3 produces at least the same number of linear polynomials.

Proposition 8. *Let C be a set of clauses, let I be the ideal generated by an algebraic representation of C, and let σ be a degree compatible term ordering.*

(1) Let $L \subset I$ be an LT_σ-interreduced set of linear polynomials. Let G be the reduced Boolean σ-Gröbner basis of I. Then we have $\#L \leq \#\{g \in G \mid \deg(g) = 1\}$.

Algorithm 4 (Combinatorial Search for Linear Polynomials)
Input: A set of clauses $B = \{c_1, \ldots, c_k\}$.
Output: A set of linear polynomials L such that the ideal generated by any algebraic representation of B contains L.

$L' := \emptyset$
foreach subset $V = \{X_{i_1}, \ldots, X_{i_\ell}\}$ **of** $\mathrm{var}(B)$ **do**
  $B^+ := \emptyset$
  $B^- := \emptyset$
  for $j = 1, \ldots, k$ **do**
    $B^+ := B^+ \cup B^+_{j,V}$
    $B^- := B^- \cup B^-_{j,V}$
  end for
  if $\#B^+ = 2^{\ell-1}$ **then**
    $L' := L' \cup \{x_{i_1} + \cdots + x_{i_\ell} + 1\}$
  end if
  if $\#B^- = 2^{\ell-1}$ **then**
    $L' := L' \cup \{x_{i_1} + \cdots + x_{i_\ell}\}$
  end if
end foreach
Let L be an LT_σ-interreduced $\mathbb{F}_2$-basis of $\langle L' \rangle_{\mathbb{F}_2}$
return L

(2) Let $m = 2$ be an overlapping number. Let L be the output of Algorithm4(C). Let S be the output of Algorithm3(C, σ, m). Then we have $\#L \leq \#\{s \in S \mid \deg(s) = 1\}$.

Proof. (1) Let $f \in I = \langle G \rangle$ be a linear polynomial. If $\mathrm{LT}_\sigma(f) \in I$, then $\mathrm{LT}_\sigma(f) \in G$ and we are done. In the other case, we know that $\mathrm{LT}_\sigma(f)$ is minimal in $\mathrm{LT}_\sigma(I)$ with respect to divisibility. The tail of f can be reduced only by linear polynomials, and this results in a linear polynomial again. After all reductions we still have a linear polynomial in G.

(2) Assume that Algorithm4(C) has discovered a set of clauses K which contains a complete signed block after extension. Using Algorithm2(C, m), compute a pair $(\mathcal{B}, T)$. If $\#\,\mathrm{var}(K) = 1$, then the corresponding linear polynomial will be derived from a clause in T and we are done. If $\#\,\mathrm{var}(K) \geq 2$, then K will appear in one $B \in \mathcal{B}$, and we can use (1). □

During our experiments, we found that conversion of a Boolean system to CNF and back to ANF may give us enough linearly independent linear polynomials to solve the initial system S. Note that having n linearly independent linear polynomials in the ideal $\langle S \rangle \subset \mathbb{B}_n$ is enough to derive the unique solution of the system by Gaußian elimination. We observed this behavior for the polynomials representing Small-scale AES encryption `AES-1-1-1-4` and `AES-2-1-1-4`. The polynomials can be found in Sage [21]. For bigger examples, this is usually not the case. On the other hand, one frequently gains additional linear polynomials in the ideal by this technique. It is made explicit in Algorithm 5.

Algorithm 5 (Generating Linear Polynomials in the Ideal)
Input: A set of Boolean polynomials S, a degree compatible term ordering σ, and an overlapping number $m \in \mathbb{N}$.
Output: A set of linear polynomials in $\langle S \rangle$.
Requires: Algorithm 3

1: Compute a logical representation of S by a sparse conversion method. Call the result C.
2: $Q :=$ Algorithm3(C, σ, m)
3: $L := \{l \in Q \mid \deg(l) = 1\}$
4: **return** L

7 Experiments

In this section we examine the efficiency of the proposed improvements of the CNF to ANF conversion. In Table 1 we compare Algorithm 1 with Algorithm 3 w.r.t. the degree of the resulting algebraic representations.

The tests have been executed on a compute server having a 3.00 GHz Intel(R) Xeon(R) CPU E5-2623 v3 and a total of 48 GB RAM. All algorithms in this paper were prototypically implemented in `python` v2.7 using the `PolyBoRi` library [6] integrated in `Sage` [21] for the Gröbner basis computations. We choose the specific mid-size instances from the following benchmark suites: the logical representations of the encryption of the Small-scale AES cipher [13] and factoring of integers [4]. Instances of type `AES-`a`-`b`-`c`-`d represent propositional formulae in CNF derived from the gate level circuit implementation of the Small-scale AES with a rounds, the state matrix of size $b \times c$ and d-bit words in each state cell. Instances of type `fact-`a`-`b represent the problem of factoring the product $a \cdot b$ for the given two primes a, b.

Table 1 provides information about the number of variables and the number of clauses contained in the CNF instance, as well as the total number of linear, quadratic and higher degree (i.e., greater than 2) polynomials produced by Algorithms 1 and 3. We use $\sigma =$ `degrevlex` and $m = 2$. The later parameter performed the best (see Proposition 8), because it does not create big blocks B_c in Algorithm 2 that are too hard for the Gröbner basis computation. On the other hand, choosing $m \geq 3$ does not make sense in most examples, because the CNF instances usually contain many clauses of length 3.

From the results in Table 1 we clearly see that the algebraic representation given by Algorithm 3 produces lower degree polynomials than the one by Algorithm 1. While Algorithm 1 usually produces only very few linear polynomials, the table shows that Algorithm 3 tends to return enough linear polynomials to eliminate approximately one third of all indeterminates. Moreover, we note that Algorithm 3 almost completely avoided to produce polynomials of degree ≥ 3.

In our experiments, the computation of the reduced Gröbner bases of the conversions of the block B_c did not pose problems. If necessary, one could calculate them in parallel. Both algorithms need extra time for the setup of the Boolean rings. This could be a problem for larger CNFs having thousands of variables. It

Instance	CNF		Algorithm 1			Algorithm 3		
	#vars	#clauses	#lin	#quad	#high	#lin	#quad	#high
AES-10-1-2-4	1081	3361	1	1792	1568	337	2194	0
AES-10-1-4-4	1862	5824	1	2986	2837	604	3692	0
AES-10-2-2-4	2441	7841	1	3584	4256	947	4407	0
AES-10-2-4-4	4289	13904	1	5986	7917	1785	7353	0
AES-10-4-1-4	3149	10065	1	4800	5264	1149	5915	0
AES-2-1-2-4	237	701	1	360	340	70	453	0
AES-2-1-4-4	412	1218	1	598	619	132	746	0
AES-2-2-2-4	526	1615	1	716	898	201	882	0
AES-2-2-4-4	935	2883	1	1196	1686	375	1491	0
AES-2-4-1-4	669	2065	1	960	1104	241	1191	0
AES-2-4-2-4	1157	3652	1	1434	2217	501	1778	0
AES-2-4-4-4	2077	6596	1	2394	4201	957	2978	0
fact-12601-18701	745	3853	2	616	3235	291	1365	2
fact-151-283	271	1333	2	250	1081	115	471	2
fact-1777-491	403	2029	2	354	1673	166	713	2
fact-2393-3371	466	2380	2	400	1978	181	855	2
fact-373-929	328	1640	2	294	1344	131	593	2
fact-583909-600203	1280	6784	2	1010	5772	471	2428	2
fact-59-1009	328	1640	2	294	1344	149	544	2
fact-59441-62201	826	4312	2	676	3634	318	1527	2
fact-81551-100057	947	4945	2	770	4173	359	1767	2
fact-9601-10067	638	3296	2	532	2762	243	1188	2

Table 1. (Number of converted polynomials by degree.)

can be overcome by defining local Boolean rings for each B_c and then rewriting the local variables in B_c to their global names. Note that $\#\operatorname{var}(B_c)$ tends to be much smaller than $\#\operatorname{var}(C)$. Moreover, caching of the standard representations of short clauses is possible. Polynomials of type $\prod_{i=1}^{l}(x_i + a_i)$ can be precomputed and stored in ANF for small values of ℓ. Then the corresponding values $a_i \in \mathbb{F}_2$ are substituted for a given clause.

Proposition 4 states that Algorithm 3 does simple logical reasoning, e.g., it applies the resolution rule to certain subformulae. Thus it tends to produce lower degree polynomials. One possible enhancement would be to run a SAT solver on a given set of clauses C for a while, and then to apply Algorithm 3 on C together with the newly found clauses (e.g., conflict clauses). We believe that this would produce even more low degree polynomials.

Acknowledgments. The authors thank Martin Albrecht and Alexander Dreyer for providing us with a better insight into the structure of `Sage` and `PolyBoRi`, as well as Mate Soos for helpful discussions about the implementation of SAT solvers. This work was financially supported by the DFG project "Algebraische Fehlerangriffe" [KR 1907/6-1].

References

1. Audemard, G., and Simon, L. Glucose in the SAT 2014 Competition. In *SAT Competition 2014: Solver and Benchmark Descriptions* (2014), Univ. of Helsinki, pp. 31–32.
2. Bard, G. *Algebraic Cryptanalysis*. Springer, Heidelberg, 2009.
3. Bard, G. V., Courtois, N. T., and Jefferson, C. Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over GF(2) via SAT-solvers. IACR Cryptology ePrint Archive.
4. Bebel, J., and Yuen, H. Hard SAT instances based on factoring. In *SAT Competition 2013: Solver and Benchmark Descriptions* (2013), Univ. of Helsinki, p. 102.
5. Brickenstein, M. *Boolean Gröbner Bases: Theory, Algorithms and Applications*. Logos Verlag, Berlin, 2010.
6. Brickenstein, M., and Dreyer, A. PolyBoRi: a framework for Gröbner basis computations with Boolean polynomials. *J. Symbolic Comput. 44* (2009), 1326–1345.
7. Condrat, C., and Kalla, P. A Gröbner basis approach to CNF-formulae preprocessing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2007), Springer, pp. 618–631.
8. Courtois, N. T., and Patarin, J. About the XL algorithm over GF(2). In *Cryptographers Track at the RSA Conference* (2003), Springer, pp. 141–157.
9. Courtois, N. T., Sepehrdad, P., Sušil, P., and Vaudenay, S. ElimLin algorithm revisited. In *Fast Software Encryption* (2012), Springer, pp. 306–325.
10. Dreyer, A., and Nguyen, T. H. Improving Gröbner-based clause learning for SAT solving industrial sized Boolean problems. In *Young Researcher Symposium (YRS)* (Kaiserslautern, 2013), Fraunhofer ITWM, pp. 72–77.
11. Een, N., and Sörensson, N. *MiniSat: A SAT solver with conflict-clause minimization*, 2005. see `http://minisat.se`.
12. Faugère, J.-C. FGb: a library for computing Gröbner bases. In *International Congress on Mathematical Software* (2010), Springer, pp. 84–87.
13. Gay, M., Burchard, J., Horáček, J., Messeng Ekossono, A. S., Schubert, T., Becker, B., Kreuzer, M., and Polian, I. Small scale AES toolbox: algebraic and propositional formulas, circuit-implementations and fault equations. In *Conf. on Trustworthy Manufacturing and Utilization of Secure Devices (TRUDEVICE 2016)* (Barcelona, 2016).
14. Horáček, J., Kreuzer, M., and Messeng Ekossono, A. S. Computing Boolean border bases. In *18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC* (Timisoara, 2016), IEEE, pp. 465–472.
15. Hsiang, J. Refutational theorem proving using term-rewriting systems. *Artif. Intell. 25* (1985), 255–300.
16. Jovanovic, P., and Kreuzer, M. Algebraic attacks using SAT-solvers. *Groups Complex. Cryptol. 2* (2010), 247–259.
17. Kreuzer, M., and Robbiano, L. *Computational Commutative Algebra 1*. Springer, Heidelberg, 2000.
18. Schubert, T., and Reimer, S. *antom*, 2016. see `https://projects.informatik.uni-freiburg.de/projects/antom`.
19. Soos, M. *CryptoMiniSat SAT solver v5.0.1*, 2016. see `http://www.msoos.org`.

20. The ApCoCoA Team. *ApCoCoA: Applied Computations in Commutative Algebra*. available at `http://apcocoa.uni-passau.de`.
21. The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 7.5.1)*, 2017. see `http://www.sagemath.org`.
22. Zengler, C., and Küchlin, W. Extending clause learning of SAT solvers with Boolean Gröbner bases. In *International Workshop on Computer Algebra in Scientific Computing* (2010), Springer, pp. 293–302.