DISSERTATION

Algebraic and Logic Solving Methods for Cryptanalysis

Submitted to the Faculty of Computer Science and Mathematics
of the University of Passau in Partial Fulfillment of the Requirements
for the Degree of Doctor of Natural Sciences

Jan Horáček

Advisor:	**Prof. Dr. Martin Kreuzer**	Chair of Symbolic Computation, University of Passau
External Referee:	**Prof. Dr. Armin Biere**	Institute for Formal Models and Verification, Johannes Kepler University Linz

Passau, June 2019

Abstract

Algebraic solving of polynomial systems and satisfiability of propositional logic formulas are not two completely separate research areas, as it may appear at first sight. In fact, many problems coming from cryptanalysis, such as algebraic fault attacks, can be rephrased as solving a set of Boolean polynomials or as deciding the satisfiability of a propositional logic formula. Thus one can analyze the security of cryptosystems by applying standard solving methods from computer algebra and SAT solving. This doctoral thesis is dedicated to studying solvers that are based on logic and algebra separately as well as integrating them into one such that the combined solvers become more powerful tools for cryptanalysis.

This disseration is divided into three parts. In this first part, we recall some theory and basic techniques for algebraic and logic solving. We focus mainly on DPLL-based SAT solving and techniques that are related to border bases and Gröbner bases. In particular, we describe in detail the Border Basis Algorithm and discuss its specialized version for Boolean polynomials called the Boolean Border Basis Algorithm.

In the second part of the thesis, we deal with connecting solvers based on algebra and logic. The ultimate goal is to combine the strength of different solvers into one. Namely, we fuse the XOR reasoning from algebraic solvers with the light, efficient design of SAT solvers. As a first step in this direction, we design various conversions from sets of clauses to sets of Boolean polynomials, and vice versa, such that solutions and models are preserved via the conversions. In particular, 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. The above conversions allow us to integrate both solvers via a communication interface.

To reach an even tighter integration, we consider proof systems that combine resolution and polynomial calculus, i.e. the two most used proof systems in logic and algebraic solving. Based on such a proof system, which we call SRES, we introduce new types of solving algorithms that demostrate the synergy between Gröbner-like and DPLL-like solving. At the end of the second part of the dissertation, we provide some experiments based on a new benchmark which illustrate that the our new method based on DPLL has the potential to outperform CDCL SAT solvers.

In the third part of the thesis, we focus on practical attacks on various cryptograhic primitives. For instance, we apply SAT solvers in the case of algebraic fault attacks on the symmetric ciphers LED and derivatives of the block cipher AES. The main goal there is to derive so-called fault equations automatically from the hardware description of the cryptosystem and thus automatizate the attack. To give some extra power to a SAT solver that inverts the hash functions SHA-1 and SHA-2, we describe how to tweak the SAT solver using a programmatic interface such that the propagation of the solver and thus the attack itself is improved.

Keywords: Boolean polynomial, border basis, SAT solving, combined proof system, algebraic normal form, conjunctive normal form, algebraic fault attack

Acknowledgements

First of all, I would like to thank my advisor Martin Kreuzer for supporting me on the path of academic research, for proofreading this text, and for continuously inspiring me to learn new things. He also accepted me to the DFG project "Algebraic Fault Attacks" [KR 1907/6-1(2)] where I was able to work with Jan Burchard, Maël Gay, Tobias Paxian, Ange-Salomé Messeng Ekossono, Bernd Becker, Tobias Schubert, Martin Kreuzer and Ilia Polian in a very productive atmosphere.

My special thanks go to Matthew England for introducing me to the EU project "SC-square" [H2020-FETOPEN-2015-CSA] such that I was able to meet many top researchers interested in combining computer algebra systems and SAT solving. I want to express my deep appreciation to Vijay Ganesh and his team who invited me twice to collaborate with University of Waterloo. Besides the names mentioned above, I would like to thank the following persons for many fruitful discussions over the years: Ilias Kotsireas, Saeed Nejati, Michael Brickenstein, Alexander Dreyer, Anna M. Bigatti, John Abbott, Lorenzo Robbiano, Mate Soos, Manuel Kauers, Martin Albrecht, Armin Biere, Bruno Buchberger, Christopher W. Brown, Daniel Lichtblau, Thomas Sturm, Stephen Forrest, Philipp Jovanovic, Jan Krajíček, Mikoláš Janota and many others. I would also like to thank Armin Biere for accepting to be the external referee for this thesis.

Moreover, I would like thank my colleagues and the staff at the University of Passau, in particular, our secretary, Nathalie Vollstädt, for advising me in organizational matters and our assistant at the chair, Florian Walsh, for helping me to implement some test scripts.

Finally, I deeply appreciate the care and the support from my mom, my dad, and my brother Jaroslav, who always motivate me to go on. I wish I could adequately express how much love and positive energy I have got from my girlfriend Marie. Unfortunately, there would be no space left for the thesis itself.

Contents

List of Symbols . . . v

1 Introduction . . . **1**
1.1 Motivation . . . 1
1.2 The State-of-the-Art and the Contributions of This Dissertation . . . 3
1.3 Structure and Content . . . 6

2 Background . . . **9**
2.1 Some Algebra Fundamentals . . . 9
2.2 Some Logic Fundamentals . . . 16
2.3 Algebraic Solvers . . . 19
2.4 SAT Solvers . . . 23
2.5 SMT and Programmatic SAT . . . 31
2.6 An Overview of Other Solving Techniques . . . 34
2.7 Block Ciphers and Hash Functions . . . 35
2.8 Cryptanalysis and Fault Attacks . . . 38

3 The Boolean Border Basis Algorithm . . . **41**
3.1 Border Bases . . . 42
3.2 The Border Basis Algorithms . . . 44
3.3 Squarefree Terms and Their Order Ideals . . . 46
3.4 Linear Interreduction for Boolean Polynomials . . . 49
3.5 Implementation of Boolean Polynomials and Linear Interreduction . . . 52
3.6 The BBBA Refined . . . 55
3.7 Improvements of the BBBA . . . 59
3.8 Experiments . . . 60

4 Integrating Algebraic and SAT Solving . . . **65**
4.1 Preliminaries . . . 66
4.2 Conversions from ANF to CNF . . . 67
4.3 The Standard Conversion from CNF to ANF . . . 68
4.4 A Blockwise Conversion from CNF to ANF . . . 69
4.5 Conversion to Linear Polynomials . . . 74
4.6 Some Applications of the Conversion Algorithms . . . 77
4.7 The Integration of the BBBA with a SAT Solver . . . 78

4.8 Design of the Communication 81
4.9 Modifications of the SAT Solver 82
4.10 Experiments 83

5 Proof Systems and SRES 89
5.1 Preliminaries 90
5.2 An Algebraic Extension of Resolution 96
5.3 Completeness of SRES 101
5.4 Some Example Proofs Using SRES 103
5.5 SRES Closure Algorithms 108
5.6 SRES Refutation Using DPLL Techniques 115
5.7 Experiments 118

6 Attacking AES and LED 123
6.1 Description of AES 124
6.2 Description of LED 125
6.3 An Automatic Construction of AFA 126
6.4 Experiments 127

7 Attacking SHA using Programmatic SAT Solvers 133
7.1 Preliminaries 134
7.2 Description of SHA-1 134
7.3 Description of SHA-256 135
7.4 Algebraic Fault Attacks on SHA 136
7.5 Experiments 137

Bibliography 143

Publications 153

List of Symbols

$\#$	the cardinality of a finite set
$\mathbb{N}$	the natural numbers including 0
$\mathbb{N}_+$	the natural numbers excluding 0
$\mathbb{Z}$	the ring of integers
$\mathbb{F}_2$	the field of two elements
K	a field
σ	a term ordering
$\mathbb{T}^n$	the set of terms in the indeterminates $x_1, \ldots, x_n$
$\mathbb{S}^n$	the set of squarefree terms in the indeterminates $x_1, \ldots, x_n$
$\mathcal{O}$	an order ideal in $\mathbb{T}^n$
$\langle T \rangle_{\mathrm{OI}}$	the order ideal cogenerated by T
$\partial\mathcal{O}$	the border of $\mathcal{O}$
P	a polynomial ring $K[x_1, \ldots, x_n]$
I	an ideal in P
$\mathrm{LT}_\sigma(f)$	the leading term of f w.r.t. σ
$\deg(f)$	the degree of f
$\mathrm{Supp}(f)$	the support of f
$\mathrm{Var}(f)$	the set of all indeterminates (variables) appearing in f
$\langle S \rangle$	the ideal generated by S
$\langle S \rangle_K$	the K-linear subspace generated by S
$\mathrm{CM}_\sigma(S)$	the coefficient matrix of S w.r.t. σ
$S^{\,\mathrm{sf}}$	the squarefree subset of S
$\mathbb{B}_n$	the ring of Boolean polynomials
$\mathcal{Z}(S)$	the set of $\mathbb{F}_2$-rational zeros of S
$\mathcal{S}(C)$	the set of models of C
$\bar{\ }$, $\wedge$, $\vee$, $\oplus$	logical `NOT`, `AND`, `OR`, and `XOR`

Chapter 1

Introduction

1.1 Motivation

The theory and techniques described in this doctoral thesis are motivated by solving hard problems coming from cryptology. *Cryptography* is a research field whose main goals are to study and design cryptographic primitives (i.e. low-level, basic cryptographic algorithms) such as block ciphers, stream ciphers, one-way functions, hash functions, etc., as well as to integrate several cryptographic primitives into cryptographic protocols such as key agreement methods, digital cash, secure multiparty computation, e-voting, etc. *Cryptanalysis* is a research area whose main objectives are to analyze and conduct attacks on cryptographic algorithms. Typically, an attacker wants to recover the secret key or an inner state of the cryptosystem. Both fields can be seen as one discipline, *cryptology*, because cryptographers must have a good overview of attacks when designing new protocols, and cryptanalysts set new standards for cryptography by new attack techniques. For background for cryptology, we refer to the book [108].

Cryptographic algorithms have frequently special features. For instance, many block ciphers are `XOR`-rich (i.e. their hardware implementations contain a lot of `XOR` gates), they typically adhere to an iterative design using only a few basic transformations, and they are carefully taylored to be effective in hardware, software, or both. The `XOR` constraints are usually used for combining secret information with the internal state of a cryptosystem. A careful analysis of this operation goes back to the work of Shannon, and it can be shown that this operation may provide so-called perfect secrecy (see [108, Sect. 2.3]), i.e. it has desired cryptographic properties.

From the theoretic point of view, cryptographic transformations are nothing more than Boolean maps (i.e. maps $\mathbb{F}_2^n \to \mathbb{F}_2^m$ where $\mathbb{F}_2$ is the field of two elements and $n, m \in \mathbb{N}$) that transform input bits to output bits, and thus they can be expressed by Boolean polynomials or by propositional formulae. Using the above representations, it is possible to analyze the attacks on ciphers via algebraic or logic methods and tools. The most straightforward approach to cryptanalysis using algebraic representations is given by *algebraic attacks* (see [10]). The idea is to represent an entire cipher as a set of Boolean polynomials or as a propositional formula and to derive secret bits by standard solving methods of algebra or logic. Countermeasures against algebraic attacks have become a standard for designing many symmetric cryptosystems. Hence easy algebraic relations of low-degree complexity should preferably not exist in modern ciphers. However, the connection between a convenient algebraic representation and security remains unclear. For instance, the S-box of the block cipher AES has a very nice, compact algebraic structure, but it is not known how to use this fact for an actual attack.

A *fault attack* is a special kind of *side-channel attack* (i.e. an attack based on informa-

tion gained from the implementation) where the attacker has access to the (hardware) implementation of the function and is able to intentionally inject physical disturbances during the operation. *Algebraic fault attacks* (often abbreviated to AFA) combine algebraic and fault attacks. The main idea of AFA is to describe so-called fault equations (i.e. equations which describe the fault propagation) using Boolean polynomials. Such polynomials are appended to an algebraic representation of the cipher, and the resulting system is solved for the secret key. For further reading on fault attacks, we refer to [12].

In the case of attacks which rely on an algebraic description of the cipher, we end up with a system of Boolean polynomials that we would like to solve (or a Boolean formula whose models we would like to determine). There exists a great variety of algorithms coming from different areas such as commutative algebra, SAT or SMT, that can be used to tackle these instances. In this doctoral thesis, we focus on algebraic and SAT solving.

Algebraic solving techniques require the concept of Boolean polynomials, i.e. the ring of Boolean polynomials $\mathbb{B}_n = P/F$ with the polynomial ring $P = \mathbb{F}_2[x_1, \ldots, x_n]$ over the field $\mathbb{F}_2$ of two elements and $F = \langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$. The above setting is very convenient because we search for 0/1-solutions, i.e. we want to describe the set

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

for a set $S \subseteq \mathbb{B}_n$. Solvers that search for more suitable generators of a given Boolean ideal are referred to as *algebraic solvers*. Examples of algebraic solvers are the Gröbner Basis Algorithm, the Border Basis Algorithm, the XL algorithm, etc. The basic principle of algebraic solvers is to generate new polynomials in the ideal and simplify the newly derived polynomials by the old ones in order to find new leading terms. For an overview of algebraic solving techniques, see [10, Ch. 12].

Typically, logic solving requires propositional formulas of the shape

$$\varphi = (L_{1,1} \vee \cdots \vee L_{1,n_1}) \wedge \cdots \wedge (L_{k,1} \vee \cdots \vee L_{k,n_k}),$$

with literals $L_{i,j}$, i.e. $L_{i,j}$ is a logical variable X_i or its negation $\bar{X}_i$. The task for *SAT solvers* is to find an element in the set

$$\mathcal{S}(\varphi) = \big\{a \in \{0,1\}^n \mid \varphi(a) = 1\big\}.$$

Many SAT solvers used in practise perform depth-first search and are based on DPLL. Most of these solvers rely on a *conflict-driven conflict-learning* (CDCL) procedure, i.e. on DPLL with the addition of clause learning. The CDCL solvers generate new clauses, called learned clauses, that guide the computation. For a detailed background on SAT solvers, see [17].

Note that the algebraic and logic solvers infer new constraints in a very different way which seem to be complementary to each other. For instance, `XOR` constraints are handled very differently in SAT solvers and in computer algebra systems. Even though algebraic solvers support `XOR` reasoning due to addition over $\mathbb{F}_2$, they do not in general perform very well on cryptographic instances. Thus a natural next step is to *integrate* algebraic and SAT solving to combine the strengths of both solvers in one.

In principle, this is a task for finding suitable *ANF to CNF conversions* (such as one in [11]), and vice versa, between different syntaxes such that their solutions (zeros of polynomials and models of propositional formulas) are preserved. In this way, it is possible to design a portfolio solver combining algebraic and logic techniques.

To push our understanding of the solvers even further, it is very handy to consider the solving techniques as procedures for generating proofs in a proof system. For theory on proof systems, see [71]. This allows designing *combined proof systems* that synthesize *polynomial calculus* and *resolution.* Algebraic solvers use polynomial calculus, i.e. the main inference rules are given as follows.

$$\frac{f \qquad g}{f+g} \qquad\qquad \frac{f}{x_i f}$$

where f, g are Boolean polynomials in $\mathbb{B}_n$ and x_i is an indeterminate. Resolution is probably the most used inference rule in SAT solvers (e.g., the learned clauses in the CDCL procedure are resolvents of the input formula). It corresponds to the rule of inference

$$\frac{c \cup \{X_i\} \qquad c' \cup \{\bar{X}_i\}}{c \cup c'}$$

where c, c' are clauses and X_i is a logical variable such that neither X_i nor $\bar{X}_i$ are contained in $c \cup c'$. The first task towards a combined proof system is to merge syntaxes of polynomial calculus and resolution. Because products of linear polynomials of the form x_i or x_i+1 correspond to clauses (see [60]), products of arbitrary linear polynomials over $\mathbb{F}_2$ is a natural choice. This syntax corresponds to the notion of *linear clauses* in logic. Based on this syntax, we have two more challenges, more precisely, to design convenient rules of inference that operates with this structure and to implement solvers that are based on the combined proof system.

1.2 The State-of-the-Art and the Contributions of This Dissertation

In this section we sum up some questions we ask in this doctoral thesis and describe the answers we provide. Because the questions are general, we give a case study (i.e. on which subcase of the general question we focus on) for each question. Nevertheless, we believe that the techniques we use and the principles taylored for a particular application can be applied elsewhere. Moreover, we give an overview of state-of-the-art and related work which is relevant to our contributions.

How to design and implement algebraic solvers for cryptanalysis?

Case study: the Boolean Border Basis Algorithm (BBBA).

From the historical point of view, first remarks how to compute border bases were given in [83]. A complete description followed in [70]. However, the order ideals in [70]

are restricted to a special form and depend on a term ordering. For a computation of border bases that does not depend on any term ordering, we refer the reader to [67]. The characterizations of border bases were given in [69]. The only non-high-level implementation of the BBA known to the author is the one in ApCoCoA [109].

In Chapter 3 we give a detailed description of the Border Basis Algorithm (BBA) and several improvements how to enhance this algorithm for cryptanalytic use. In particular, we introduce the Boolean Border Basis Algorithm (BBBA) which is taylored to Boolean polynomials. We provide details on how to design suitable data structures used in the BBBA (e.g. coefficient matrices, terms, order ideals, etc.). The BBBA overcomes some problems of the Gröbner Basis Algorithm [72, Ch. 2] (GBA). For instance, the degree is not increased as greatly as during the GBA, and only the linear reductions are used in the BBA, unlike the normal remainder reductions in the GBA. Altogether, we introduce the first specialized implementation of the Boolean Border Basis Algorithm in C++.

How to integrate algebraic and SAT solving?

Case study: an integration of the Boolean Border Basis Algorithm (BBBA) with the SAT solver antom [101].

For an integration of SAT solvers with the Gröbner basis algorithm, we refer to [114], [87] or [37]. A combination of a SAT solver with the XL algorithm can be found in [30]. For various methods dealing with XOR clauses, i.e. linear polynomials, we refer the reader to [13, 75] or to [106] for a concrete implementation.

In Chapter 4 we describe an integration of the Boolean border basis solver and a SAT solver via communication interface in a portfolio fashion. A special subproblem of this question we deal with is to find suitable conversions between sets of clauses (CNF) and sets of Boolean polynomials in ANF. Whereas the reverse conversion has been studied before (see [11]), the CNF to ANF conversion has been achieved predominantly via a standard method in [60] 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. The resulting integration using conversions is the first framework for combining a border basis solver and a SAT solver in the literature. Furthermore, we provide examples where the integration outperforms the base solvers.

How to design combined proof systems and solvers based on them?

Case study: the proof system SRES which uses the syntax consisting of linear clauses and the rule of inference sres.

Let us mention some previous contributions to the topic of combining the resolution calculus and the polynomial calculus into a new proof system, and compare them to s-resolution. The proof system RES-LIN in [61] admits linear clauses, which correspond to our linearly split polynomials, but uses only 1-resolution steps, whereas the proof system in [13] works only for XOR clauses (i.e., linear Boolean polynomials) and relies on Gaußian elimination. Moreover, the proof system RLIN in [96] applies a more general addition

of linear factors than s-resolution and does not target the main idea of S-polynomials, namely cancellation of leading terms. Apparently, these proof systems have not led to efficient implementations. Moreover, we are not aware of any actual implementation of a solver for sets of linear clauses.

In Chapter 5 we study a combination of resolution and polynomial calculus. Using a syntax allowing products of linear polynomials that correspond to linear clauses, we taylor a special rule of inference, called s-resolution (sres), and a new proof system SRES. Because of its strong algebraic background, we call sres an algebraic extension of resolution. Our main goal here is to lift up the concepts known from SAT solving of sets of clauses to solving of set of linear clauses and to implement solvers based on SRES. In particular, we describe a closure and DPLL-based algorithm which use SRES. We implement both those algorithms in `python`. Furthermore, we provide a description of a benchmark which is hard for resolution-based SAT solvers but has short proofs in SRES.

What about breaking real-world cryptographic primitives?

Case study: algebraic fault attacks on LED and small-scale AES using the tool `AutoFault`, algebraic fault attacks on SHA-1 and SHA-2 using a programmatic SAT solver based on `MapleSAT` [80].

The standard reference for hand-crafting fault equations for AES and LED is [65,112]. A framework for automatically creating fault equations can be found in [115]. The secret is determined by solving the set of constraints describing the cryptosystem under attack, i.e. an algebraic representation has to be created. Countermeasures against fault attacks include low-level approaches like adding shields to cover metallization levels [77], placing sensors in the circuitry [95], or higher-level methods such as error-detecting codes [66].

The initial work on fault attacks on the SHA family goes back as far as [78], where a differential fault attack is applied to SHACAL-1 (i.e. a block cipher adopting the structure of SHA-1). Authors of [51] extended the attack to SHA-1. Because the structure of SHA-1 is more difficult than SHACAL-1, they needed more than a thousand fault injections to derive a message.

In Chapter 6 we discuss fault attacks on scaled variants of the block ciphers AES and the lightweight cipher LED. Instead of crafting fault equations manually, we introduce the tool `AutoFault` for creating automatic fault equations. To push the automation even further, the equations are derived automatically from the hardware description of the cryptosystem. Moreover, we use various new ways how algebraic fault attacks (AFA) can be encoded.

In contrast to [115], `AutoFault` uses hardware descriptions which are usually publicly available. To the best of our knowledge, `AutoFault` is the first approach to breaking LED-64 using no “manual” cryptanalytic information (that is, without manually derived constraints beyond the circuit or the fault descriptions). Comparing `AutoFault` to statistical approaches, e.g. in [48, 79], a successful AFA requires less fault injections, but of higher precision.

In Chapter 7 we introduce a new approach to AFA using programmatic SAT solvers.

The programmatic interface is used to strengthen the performance of the base solver on the PREIMAGE problem of the SHA families. While encoding SHA into CNF in the similar manner as in [91], we observed that the resulting CNF does not perform well with the respect to the native Boolean Constraint Propagation (BCP) of the SAT solver `MapleSAT` [80]. This problem is captured by the notion of a special version of general arc consistency (GAC).

The propagation can be easily enhanced via the programmatic interface. Moreover, the programmatic interface can implement other features of the attack (e.g. checking of message candidates, etc.) that has to be taken care of otherwise from the "outside" of the solver. Our attack via the programmatic interface outperforms the approaches in [62], [51] and [50] both in terms of the number of fault injections and solving time.

1.3 Structure and Content

Every chapter in this dissertation starts with a motivation, an overview of related work and its content. Because of this, the chapters can be read as standalone parts with a few references to Chapter 2. The main chapters (i.e. all excluding the first two) finish with experiments that illustrate the performance of the implementations. In the following paragraphs, we outline the structure and the content of each chapter.

In Chapter 2 we recall some fundamentals from algebra and logic. Moreover, we give a small introduction to side-channel cryptanalysis.

In the beginning of Chapter 3 we give the definition of border bases. In its further sections we focus on the theory regarding the Boolean Border Basis Algorithm and deal with implementation aspects of this algorithm. This chapter is based on [53, 55, 58, 59].

Chapter 4 is divided into two parts. In the first part, we recall various conversions between ANF and CNF. In particular, we introduce a new blockwise algorithm for the CNF to ANF conversion. In the second part, we use these conversions for an integration of algebraic and SAT solving, more precisely, for an integration of the border basis solver with a CDCL SAT solver via a communication interface. This chapter is based on the articles [53–55].

In Chapter 5 we start with some fundamentals from the theory of proof systems and define a new proof system SRES. After we prove that its rules of inference are sound and SRES is implicationally and refutationally complete, we turn our attention to solving methods based on SRES. We focus on closure algorithms and algorithms that are based on DPLL. This chapter is based on [56, 57].

In Chapter 6 we give a brief overview of the block ciphers AES, small-scale AES (ssAES), and LED. Then we explain how to apply a practical AFA using our new tool `AutoFault` to these ciphers. This chapter is based on [28, 29, 46].

In Chapter 7 we recall some parts from the theory of constraint satisfaction problems. After describing SHA-1 and SHA-2 hash functions, we introduce an inversion attack using programmatic solvers. Specifically, we explain its two basic components: the programmatic propagator and the programmatic conflict analyzer. This chapter is based on [84].

At the end of this doctoral thesis, there is a list of all our publications relevant to

this dissertation. The results in this doctoral thesis were presented in the following conferences and workshops [1]: ICMS, FLoC, CP, MACIS, SYNASC, ISSAC, FDTC, TRUDEVICE, IVSW, CAI. Article [57] has been accepted for publication by the Journal of Symbolic Computation.

[1] If a workshop was a part of a larger conference, we list only the conference to which the workshop was affiliated.

Chapter 2

Background

Motivation In this chapter we point out basic definitions and notions from algebra, logic and cryptography that are going to be used throughout the thesis. The emphasis is put on recalling some techniques from algebraic and SAT solving. For this purpose, we need a sufficient theoretical background in the first subsections. Basically, algebraic techniques over $\mathbb{F}_2$ require the concept of Boolean polynomials and are based on rewriting polynomials and saturation of an ideal. In contrast to this approach, CDCL SAT solvers are based on "intelligent" depth-first search. Both approaches have pros and cons. On one hand, *algebraic solvers* provide native `XOR` reasoning due to addition over $\mathbb{F}_2$, solve harder problems than SAT (namely describing the whole set of solutions), but are usually slower than SAT solvers, and suffer from a higher space complexity. On the other hand, *logic solvers* do not typically support `XOR`, are usually fast (because they search only for one solution at a time) and space efficient. Thus, this chapter enables us later to study the solvers (or their combinations) and to apply them in fault injection attacks.

Literature The notations and definitions regarding commutative algebra adhere to the book [72]. The theory on border bases is taken from [73, Sect. 6.4] and [70]. The sections about Boolean functions and logic background are mostly inspired by [35]. The part about Boolean polynomials follows [22], whereas the part about SAT solving is based on [17]. The final sections about cryptology are along lines of [108]. We allow us to add a certain degree of flexibility, i.e. by modifying some algorithms or concepts slightly, such that the text gives us a stable background for the next chapters. Further related literature is given in each section separately.

Structure and contents First of all, we define basic "vocabularies" for algebra and logic. In the next step, we look into the underlying ideas behind various solving techniques. The chapter is concluded by a small introduction to block ciphers, hash functions, and side-channel cryptanalysis.

2.1 Some Algebra Fundamentals

In this section we recall basic definitions and known results from commutative algebra and introduce useful notation. The terminology follows to the notation given in [72]. We start with basic definitions regarding terms.

Definition 2.1. (a) The set $\mathbb{T}^n = \{x_1^{\alpha_1} \cdots x_n^{\alpha_n} \mid \alpha_i \geq 0\}$ is called the **set of terms** in the indeterminates $x_1, \ldots, x_n$.

(b) The set of **squarefree terms** in the indeterminates $x_1, \dots, x_n$ is defined as the set $\mathbb{S}^n = \{x_1^{\alpha_1} \cdots x_n^{\alpha_n} \mid \alpha_i \in \{0,1\}\}$.

(c) Given a term $t = x_1^{\alpha_1} \cdots x_n^{\alpha_n} \in \mathbb{T}^n$, the tuple $(\alpha_1, \dots, \alpha_n) \in \mathbb{N}^n$ is called the **exponent vector** of the term t.

(d) The **degree** of a term $t = x_1^{\alpha_1} \cdots x_n^{\alpha_n} \in \mathbb{T}^n$, denoted by $\deg(t)$, is defined as $\deg(t) = \alpha_1 + \cdots + \alpha_n$. We set $\deg(0) = -1$.

Zero exponents are often omitted in power products. For instance, we write x_1x_2 instead of $x_1^1x_2^1x_3^0$, etc. We order terms in $\mathbb{T}^n$ and $\mathbb{S}^n$ by a term ordering σ which is defined as follows.

Definition 2.2. Let $\sigma \subset \mathbb{T}^n \times \mathbb{T}^n$ be a binary relation on $\mathbb{T}^n$. We write $t_1 \geq_\sigma t_2$ if $(t_1, t_2) \in \sigma$.

(a) The relation σ is called a **term ordering** if the following conditions are satisfied for all $t_1, t_2, t_3 \in \mathbb{T}^n$.

   (i) $t_1 \geq_\sigma t_2$ or $t_2 \geq_\sigma t_1$

   (ii) $t_1 \geq_\sigma t_1$

   (iii) $t_1 \geq_\sigma t_2$ and $t_2 \geq_\sigma t_1$ imply $t_1 = t_2$

   (iv) $t_1 \geq_\sigma t_2$ and $t_2 \geq_\sigma t_3$ imply $t_1 \geq_\sigma t_3$

   (v) $t_1 \geq_\sigma t_2$ implies $t_1t_3 \geq_\sigma t_2t_3$

   (vi) $t \geq_\sigma 1$ for all $t \in \mathbb{T}^n$

(b) The relation σ is called **degree compatible** if $t_1 \geq_\sigma t_2$ for $t_1, t_2 \in \mathbb{T}^n$ implies $\deg(t_1) \geq \deg(t_2)$.

Similar interpretations are used for "$>_\sigma$", "$<_\sigma$" and "$\leq_\sigma$", e.g. $t_1 >_\sigma t_2$ for $t_1, t_2 \in \mathbb{T}^n$ means $t_1 \geq_\sigma t_2$ and $t_1 \neq t_2$, etc. The most useful term orderings are the *lexicographic ordering* `Lex`, the *degree-lexicographic ordering* `DegLex` and the *degree-reverse-lexicographic ordering* `DegRevLex`. Their definitions can be found in [72, Sect. 1.4]. Now we consider the following special sets of terms that are closed under division. As usual, we define $aT = \{a \cdot t \mid t \in T\}$ for $T \subset \mathbb{T}^n$ and $a \in \mathbb{T}^n$, and $a \mid b$ denotes divisibility of b by a.

Definition 2.3. (a) A finite (non-empty) subset $\mathcal{O}$ of $\mathbb{T}^n$ is called an **order ideal** if $t \in \mathcal{O}$ and $t' \mid t$ for $t' \in \mathbb{T}^n$ imply $t' \in \mathcal{O}$.

(b) Let $\mathcal{O}$ be an order ideal in $\mathbb{T}^n$. A set of terms $C = \{t_1, \dots, t_k\} \subseteq \mathcal{O}$ is called a set of **cogenerators** of $\mathcal{O}$ (or we say that C **cogenerates** $\mathcal{O}$) if every term in $\mathcal{O}$ divides at least one of the terms $t_1, \dots, t_k$. In this case we write $\langle C \rangle_{\mathrm{OI}} = \mathcal{O}$.

(c) A set of cogenerators $\{t_1, \dots, t_k\}$ of an order ideal is called **minimal** if no term t_i divides t_j for $j \neq i$.

(d) Given an order ideal $\mathcal{O}$ in $\mathbb{T}^n$, we call $\partial\mathcal{O} = (x_1\mathcal{O} \cup \cdots \cup x_n\mathcal{O}) \setminus \mathcal{O}$ the **border** of $\mathcal{O}$.

Notice that an order ideal is not the set of terms in a polynomial ideal, but an ideal in the category of partially ordered sets. Order ideals are represented by their (unique) minimal sets of cogenerators. Given a set of cogenerators C of an order ideal that is not minimal, we iteratively remove all terms $t \in C$ such that t divides t' for some term $t \neq t' \in C$. The next example illustrates an order ideal and its border in two indeterminates.

Example 2.4. Let $\mathcal{O} = \{1, x_1, x_2, x_1x_2\}$. Then $\mathcal{O}$ is an order ideal in $\mathbb{T}^2$ cogenerated by $C = \{x_1x_2\}$ with its border $\partial\mathcal{O} = \{x_1^2, x_1^2x_2, x_1x_2^2, x_2^2\}$. This order ideal and its border can be illustrated as follows.

△

Next we define the squarefree part of a set of terms.

Definition 2.5. Let S be a set of terms in $\mathbb{T}^n$.

(a) Then the set $S^{\text{sf}} = S \cap \mathbb{S}^n$ is called the **squarefree subset** of S.

(b) In particular, $(\partial\mathcal{O})^{\text{sf}}$ is called the **squarefree border** of an order ideal $\mathcal{O}$, i.e. $(\partial\mathcal{O})^{\text{sf}} = \big((\bigcup_{i=1}^n x_i\mathcal{O}) \setminus \mathcal{O}\big) \cap \mathbb{S}^n$.

Notice that the squarefree subset of a set of terms can be empty.

In the following we let K be a field and $P = K[x_1, \ldots, x_n]$ a polynomial ring over K. Next we recall some definitions regarding polynomials.

Definition 2.6. Let σ be a term ordering in $\mathbb{T}^n$. Let $f \in P$ be a non-zero polynomial of the form $f = c_1t_1 + \cdots + c_kf_k$ with $t_i \in \mathbb{T}^n$ and $c_i \in K \setminus \{0\}$ such that $t_1 >_\sigma t_2 >_\sigma \cdots >_\sigma t_k$.

(a) The **support** of f is defined as the set $\text{Supp}(f) = \{t_1, \ldots, t_k\} \subset \mathbb{T}^n$.

(b) The polynomial f is said to have **squarefree support** if $\text{Supp}(f) \subset \mathbb{S}^n$.

(c) The **leading term** of f is the largest term in the support of f w.r.t. σ. We write $\text{LT}_\sigma(f) = t_1$.

(d) The **degree** of f is defined by $\deg(f) = \max\{\deg(t) \mid t \in \text{Supp}(f)\}$.

(e) A set of polynomials $G \subset P$ is called **(linearly)** LT_σ**-interreduced** if $\text{LT}_\sigma(g) \neq \text{LT}_\sigma(g')$ for all $g, g' \in G$ with $g \neq g'$.

(f) The set $\mathrm{Var}(f)$ is defined as the set of all indeterminates appearing in $\mathrm{Supp}(f)$. It is called the **set of indeterminates** of f.

(g) We write $\mathrm{Supp}(G) = \bigcup_{g\in G} \mathrm{Supp}(g)$ and $\mathrm{Var}(G) = \bigcup_{g\in G} \mathrm{Var}(g)$ for a finite set of polynomials $G \subset P$.

Given a term ordering, a finite set of polynomials can be represented by a matrix as follows.

Definition 2.7. Let $G = (f_1, \ldots, f_s) \in P^s$ be a tuple of polynomials. Let $\mathrm{Supp}(G) = \{t_k, \ldots, t_1\}$ such that $t_k >_\sigma \cdots >_\sigma t_1$. The **coefficient matrix** of G w.r.t. the term ordering σ is a matrix in $K^{s\times k}$ denoted by $\mathrm{CM}_\sigma(G)$ and defined by

$$(\mathrm{CM}_\sigma(G))_{i,j} = \begin{cases} \text{the coefficient of } t_j \text{ in } f_i & \text{if } t_j \in \mathrm{Supp}(f_i), \\ 0 & \text{otherwise.} \end{cases}$$

When we speak of coefficient matrices w.r.t. sets of polynomials (and not tuples as above), we do not require any special order of the rows of the coefficient matrix. Given a tuple of polynomials $G \subseteq P$, we can linearly LT_σ-interreduce G via Gaußian elimination on the coefficient matrix of G. Thus, the coefficient matrix of a set of polynomials G is in *row echelon form* (REF) if and only if G is linearly LT_σ-interreduced.

Example 2.8. Let $\sigma = \mathtt{DegLex}$ and $P = \mathbb{Q}[x_1, x_2, x_3]$. The polynomials $G = (f_1, \ldots, f_5)$ given on the right-hand side correspond to the matrix on the left.

$$\begin{array}{c} \begin{matrix} x_1x_2x_3 & x_1x_2 & x_1 & x_2 & 1 \end{matrix} \\ \begin{pmatrix} 3 & 1 & 1 & 2 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 4 & 1 & 0 & 1 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 & 1 \end{pmatrix} \end{array} \begin{array}{l} \leftrightarrow\ f_1 = 3x_1x_2x_3 + x_1x_2 + x_1 + 2x_2 + 1 \\ \leftrightarrow\ f_2 = x_1x_2 + 1 \\ \leftrightarrow\ f_3 = 4x_1x_2 + x_1 + 1 \\ \leftrightarrow\ f_4 = 2x_2 \\ \leftrightarrow\ f_5 = 3x_2 + 1 \end{array}$$

The linear LT-intereduction of G and a row echelon form of $\mathrm{CM}_\sigma(G)$ are given below. The third row comes from $-4f_2 + f_3$ and the fifth row is equal to $3f_4 - 2f_5$.

$$\begin{array}{c} \begin{matrix} x_1x_2x_3 & x_1x_2 & x_1 & x_2 & 1 \end{matrix} \\ \begin{pmatrix} 3 & 1 & 1 & 2 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & -3 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & -2 \end{pmatrix} \end{array} \begin{array}{l} \leftrightarrow\ 3x_1x_2x_3 + x_1x_2 + x_1 + 2x_2 + 1 \\ \leftrightarrow\ x_1x_2 + 1 \\ \leftrightarrow\ x_1 - 3 \\ \leftrightarrow\ 2x_2 \\ \leftrightarrow\ -2 \end{array}$$

$\triangle$

Next we continue with definitions and notations regarding ideals and monomial ideals.

Definition 2.9. Let $f_1, \ldots, f_s$ be polynomials in P. Let σ be a term ordering.

(a) The **ideal** in P generated by a set of polynomials $\{f_1, \ldots, f_s\} \subset P$ is denoted by $\langle f_1, \ldots, f_s \rangle$.

(b) The **K-linear subspace** of P generated by a set of polynomials $\{f_1, \ldots, f_s\} \subset P$ is denoted by $\langle f_1, \ldots, f_s \rangle_K$.

(c) An ideal generated by a set of terms is called a **monomial ideal**.

(d) For an ideal $I \subset P$, the monomial ideal $\mathrm{LT}_\sigma(I) = \langle \mathrm{LT}_\sigma(f) \mid f \in I \setminus \{0\} \rangle$ is called the **leading term ideal**.

(e) For an ideal $I \subset P$, the set $\mathcal{O}_\sigma(I)$ is defined by $\mathcal{O}_\sigma(I) = \mathbb{T}^n \setminus \mathrm{LT}_\sigma(I)$.

(f) An ideal I in P is called **0-dimensional** if P/I is a finite dimensional K-vector space.

Let us provide some background to the definition given in (f). The polynomial ring P is also a vector space over K with a basis given by all terms in $\mathbb{T}^n$. Any ideal I can be viewed as a K-vector subspace of P, and thus P/I is K-vector space as well.

In the next proposition we recall some interesting connections between order ideals and monomial ideals. The proofs can be found in [73, Sect. 6.4.A].

Proposition 2.10. *Let I be a 0-dimensional ideal in P.*

(a) The complement of an order ideal is the set of terms of a monomial ideal in P.

(b) For every term ordering σ, the set $\mathcal{O}_\sigma(I)$ is an order ideal.

(c) The residue classes of the elements in $\mathcal{O}_\sigma(I)$ form a K-vector space basis of P/I.

Let $\mathbb{F}_2 = \mathbb{Z}/2\mathbb{Z}$ be the field of two elements. In the rest of this section, we focus on the case $K = \mathbb{F}_2$, i.e. $P = \mathbb{F}_2[x_1, \ldots, x_n]$.

Definition 2.11. Let $P = \mathbb{F}_2[x_1, \ldots, x_n]$.

(a) The ideal $F = \langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$ is called the **field ideal**.

(b) The ring $\mathbb{B}_n = P/F$ called the **ring of Boolean polynomials** in the indeterminates $x_1, \ldots, x_n$.

(c) Polynomials in P with squarefree support are said to be in **algebraic normal form (ANF)**.

(d) A map $\varphi : \mathbb{F}_2^n \to \mathbb{F}_2$ is called a **Boolean function** in n variables.

(e) A map $\psi : \mathbb{F}_2^n \to \mathbb{F}_2^m$ for $m \in \mathbb{N}$ is called an **m-dimensional Boolean map** in n variables.

(f) Let $a = (a_1, \ldots, a_n) \in \mathbb{F}_2^n$ and $f \in P$. The notation $f(a)$ denotes the **evaluation** of f at the point a. In particular, $f|_{x_i \mapsto a_i}$ denotes the polynomial which is obtained by substituting $x_i \mapsto a_i$ into f.

The ideal F is called the field ideal, since it is the vanishing ideal of $\mathbb{F}_2^n$. An m-dimensional Boolean map ψ in n variables can be viewed as a collection of Boolean functions $\varphi_1, \dots, \varphi_m$ in n variables such that $\psi(a) = (\varphi_1(a), \dots, \varphi_m(a))$. Similar notations as in (f) are used for Boolean maps and functions. For a map ψ, $\psi|_{x_i \mapsto a_i}$ denotes the map obtained from ψ by fixing the coordinate $x_i = a_i$.

The following proposition contains some useful facts about Boolean polynomials and functions.

Proposition 2.12. *Let $P = \mathbb{F}_2[x_1, \dots, x_n]$. Let R be the set of all polynomials in P which are in ANF.*

(a) The set R is a set of representatives of the residue classes in $\mathbb{B}_n$. We have $\#R = 2^{2^n}$.

(b) Every ideal in $\mathbb{B}_n$ uniquely corresponds to an ideal in P containing the field ideal.

(c) Let I be an ideal in P with $F \subset I$. Then I is zero-dimensional, radical (i.e. $I = \sqrt{I} = \{f \in I \mid f^i \in I \text{ for some } i \in \mathbb{N}_+\}$), and principal (i.e. generated by one Boolean polynomial).

(d) Let φ be a Boolean function in n variables. Then there exists a unique polynomial $r \in R$ in ANF such that $\varphi(a) = r(a)$ for all $a \in \mathbb{F}_2^n$. Conversely, every polynomial $r \in R$ corresponds to a unique Boolean function in n variables.

(e) We have $f^2 = f$ in $\mathbb{B}_n$ for all $f \in \mathbb{B}_n$.

Proof. (a) Consider a residue class $g + F$ in $\mathbb{B}_n$ with $g \in P$. We divide g by F with a remainder (see [72, Th. 1.6.4]), and we get $g = h + f$ with a squarefree polynomial $h \in R$ and $f \in \langle F \rangle$. Thus, $g - h \in F$, i.e. h represents the residue class $g + F$. We have $\#\mathbb{S}^n = \binom{n}{0} + \binom{n}{1} + \dots + \binom{n}{n} = 2^n$. Because each coefficient of the squarefree term is 0 or 1, we have $\#R = 2^{2^n}$.

(b) Let us define a map α as follows. We set $\alpha(I) = \{g + F \mid g \in I\} \subset \mathbb{B}_n$ for ideals I in P with $F \subseteq I$, i.e. the domain of α is the set of all ideals in P. Note that $\alpha(I)$ is an ideal in $\mathbb{B}_n$. Conversely, we define a map β as follows $\beta(J) = \{h \in P \mid h + F \in J\} \subset P$, i.e. the domain of β is the set of ideals in $\mathbb{B}_n$. Note that $\beta(J)$ is an ideal in P, and $F \subset \beta(J)$. Now, it is sufficient to show that $\alpha \circ \beta$ and $\beta \circ \alpha$ are identities on the set of ideals in P and $\mathbb{B}_n$. We prove this for $\beta \circ \alpha$. The other identity is analogous. Let I be an ideal with $F \subset I \subset P$. Then we have $\beta(\alpha(I)) = \big\{h \in P \mid h + F \in \{g + F \mid g \in I\}\big\} = \{g \in P \mid g \in I\} = I$.

(c) Any residue class in P/I is represented by $r + I$ for some $r \in R$. Hence the dimension of the $\mathbb{F}_2$-vector space is at most 2^n. The proof that I is radical can be found in [22, Th. 2.2.4]. Every ideal in P is finitely generated by Hilbert's basis theorem, and hence an ideal in the factor ring P/F is finitely generated as well. To show that I is principal, it is sufficient to prove that $\langle f, g \rangle = \langle fg + f + g \rangle$ holds in $\mathbb{B}_n$ for $f, g \in \mathbb{B}_n$. The claim then follows by induction. The inclusion "$\supseteq$" is trivial. Conversely, we have $f(fg + f + g) = fg + f + fg = f$ and $g(fg + f + g) = fg + fg + g = g$ in $\mathbb{B}_n$ which proves the other inclusion.

(d) Let us prove the first statement by induction on n. Let φ be a Boolean function in $n = 1$ variables. Then φ is equivalent to one of these polynomials in ANF: $0, 1, x_1, x_1 + 1$. Let φ is a Boolean function in $n > 1$ variables. We show that the map φ can be decomposed to $x_1 \cdot \varphi|_{x_1 \mapsto 1} + (x_1 + 1) \cdot \varphi|_{x_1 \mapsto 0}$. Indeed, if $x_1 = 0$, then the right-hand side is $\varphi|_{x_1 \mapsto 0}$, and if $x_1 = 1$, we get $\varphi|_{x_1 \mapsto 1}$. Note that $\varphi|_{x_1 \mapsto 1}$ or $\varphi|_{x_1 \mapsto 0}$ contain at most $n - 1$ variables. Thus, the claim follows by the induction hypothesis.

For the converse implication, it suffices to consider the function defined by $a \mapsto r(a)$. The uniqueness follows from the fact that $\#R = 2^{2^n}$ is equal to the cardinality of the set of Boolean functions in n variables, as shown above.

(e) It suffices to show the equivalent equality $f(f + 1) = 0$. Let $f = t_1 + t_2 + \cdots + t_k$ with $t_i \in \mathbb{S}^n$ such that $t_1 >_\sigma \cdots >_\sigma t_k$ for some term ordering σ. We count the occurrences of products $t_i t_j$ in the expression $(t_1 + \cdots + t_k)(t_1 + \cdots + t_k + 1)$. There are two ways how to obtain t_i, namely from the multiplications $t_i t_i$ and $t_i \cdot 1$. There are two ways how to obtain $t_i t_j$ with $i \neq j$, namely from the multiplications $t_i t_j$ and $t_j t_i$. Thus, the expression equals to $2 \cdot h$ for some $h \in \mathbb{B}_n$, and henceforth it is equal to 0 in $\mathbb{B}_n$. □

Claim (a) is the reason why we identify polynomials in ANF with elements in $\mathbb{B}_n$. In view of Claim (d), we see that any Boolean function can be described by a polynomial in ANF. Thus Boolean polynomials in ANF correspond 1-1 to Boolean functions.

In the next definition we are interested in $\mathbb{F}_2$-rational zeros of a set of polynomials in P. Because we have $0^k = 0$ and $1^k = 1$ for $k \in \mathbb{N}_+$, we do arithmetics in P modulo the ideal $F = \langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$, i.e. we are in the setting of Boolean polynomials.

Definition 2.13. 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

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

By the following proposition, 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. We often omit the curly brackets and write only $\mathcal{Z}(f_1, \ldots, f_k)$.

Proposition 2.14. *Let $S = \{f_1, \ldots, f_s\} \subseteq \mathbb{B}_n$ and $I = \langle S \rangle$.*

(a) We have $\mathcal{Z}(S) = \mathcal{Z}(I)$.

(b) Let J be an ideal in $\mathbb{B}_n$. We have $I = J$ if only if $\mathcal{Z}(I) = \mathcal{Z}(J)$.

(c) We have $1 \in I$ if only if $\mathcal{Z}(I) = \emptyset$.

Proof. (a) Let us prove the inclusion "$\supseteq$". Clearly, we have $S \subseteq I$. It follows that $\mathcal{Z}(I) \subseteq \mathcal{Z}(S)$. Next, we prove "$\subseteq$". Let $a \in \mathcal{Z}(S)$ and $g \in \langle S \rangle$, i.e. $g = \sum h_i f_i$ for some $h_i \in \mathbb{B}_n$. We get $g(a) = \sum h_i(a) f_i(a) = 0$, and thus $a \in \mathcal{Z}(I)$.

(b) The implication "$\Rightarrow$" follows from Claim (a) because of $I = J = \langle S \rangle$. Let us prove the other direction "$\Leftarrow$". By Proposition 2.12(c) we write $I = \langle f \rangle$ and $J = \langle g \rangle$ with $f, g \in \mathbb{B}_n$ that are in ANF. By Claim (a) we get $\mathcal{Z}(f) = \mathcal{Z}(\langle f \rangle) = \mathcal{Z}(\langle g \rangle) = \mathcal{Z}(g)$. Define two Boolean functions $\varphi, \psi : \mathbb{F}_2^n \to \mathbb{F}_2$ by $\varphi(a) := f(a)$ and $\psi(a) := g(a)$ for $a \in \mathbb{F}_2^n$. These two functions are the same because it holds $\varphi(a) = \psi(a)$ for all $a \in \mathbb{F}_2^n$. By Prop. 2.12(d) we obtain $f = g$.

(c) The proof follows from the strong version of Hilbert's Nullstellensatz (see [22, Cor. 2.2.5]). □

Note that the only Boolean polynomial f with $\mathcal{Z}(f) = \emptyset$ is the constant polynomial 1. To the previous definition we associate the following *decision* problem.

Problem:	Polynomial system solving over $\mathbb{F}_2$ (PSS)
Input:	A set of Boolean polynomials $S \subset \mathbb{B}_n$.
Question:	Is there a zero $a \in \mathbb{F}_2^n$ such that $a \in \mathcal{Z}(S)$?

The complexity class of decision problems NP (*nondeterministic polynomial time*) is a set of decision problems whose positive solutions can be verified in polynomial time. We say that a decision problem D is in the complexity class of the NP-*hard* problems when every problem in NP can be reduced to D using a polynomial time reduction. The complexity class NP-*complete* is the intersection of the classes NP-hard and NP. For more details on complexity theory and its basic definitions, see [3].

The problem PSS is known to be NP-complete (even over any finite field, see [45, Appendix 7]). The *search* version of the problem is to find a common zero if one exists. In principle, the complexity of the search problems does not differ from the decision problems, because the search versions are solvable in polynomial time given an oracle for the decision versions.

2.2 Some Logic Fundamentals

In this section we recall basic definitions and known results from logic and introduce useful notation. The terminology adheres to the notation given in [35, 104].

Definition 2.15. (a) The **alphabet** of propositional logic consists of

- **propositional variables** (sometimes called **logical variables**) $X_1, \ldots, X_n$,
- **propositional connectives** such as logical negation "$^-$" (sometimes denoted by "$\neg$"), disjunction "$\vee$", conjunction "$\wedge$", exclusive or `XOR` (sometimes denoted by "$\oplus$"),
- parentheses and the constants `True` and `False`.

(b) **Propositional logic formulas** or **Boolean formulas** in n variables are well-formed strings over the alphabet defined in (a) by finitely many recursive application of the rules (i), (ii) and (iii).

(i) The constants `True` and `False` are propositional logic formulas.

(ii) The logical variables $X_1, \dots, X_n$ are propositional logic formulas.

(iii) If φ and ψ are propositional logical formulas, then $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \oplus \psi)$ and $(\bar{\varphi})$ are propositional logic formulas.

(c) For a propositional logical formula φ, we denote $\mathrm{Var}(\varphi)$ the set of **all logical variables** appearing in φ.

In fact, the connective `XOR` in the definition is often omitted. We mention it here because we frequently deal with `XOR` in the next chapters. 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 `True` $\equiv 1$ and `False` $\equiv 0$, e.g. we have $\bar{0} = 1$ and $\bar{1} = 0$, etc. Because the addition in $\mathbb{F}_2$ is done modulo 2, it corresponds to the `XOR` operation. We go on with the definition of an another normal form, called CNF.

Definition 2.16. (a) A **literal** is a logical variable X_i or its negation $\bar{X}_i$. A literal of form X_i is called **positive**, and a literal of form $\bar{X}_i$ is called **negative**.

(b) A **clause** is a (finite) set of literals.

(c) A clause c is called a **unit clause** if $\#c = 1$.

(d) A Boolean formula of the shape

$$\varphi = (L_{1,1} \vee \cdots \vee L_{1,n_1}) \wedge \cdots \wedge (L_{k,1} \vee \cdots \vee L_{k,n_k}),$$

with literals $L_{i,j}$ is said to be in **conjunctive normal form (CNF)**.

(e) We say that the formula φ in (d) is given by its **set of clauses**

$$C = \big\{\{L_{1,1}, \dots, L_{1,n_1}\}, \dots, \{L_{k,1}, \dots, L_{k,n_k}\}\big\}.$$

Usually, we assume that a CNF formula φ is given by its set of clauses C, and we frequently identify φ and C.

So far we have looked only at the syntax of propositional formulae. Now we turn our attention to semantics. In the following we recall some definitions related to satisfiability of a Boolean formula.

Definition 2.17. Let $\varphi, \varphi_1, \dots, \varphi_m, \psi$ be propositional formulae in the logical variables $X_1, \dots, X_n$.

(a) A map $\alpha : \{X_1, \dots, X_n\} \to \{0, 1, *\}$ is called an **assignment** of the logical variables $X_1, \dots, X_n$. The symbol "$*$" denotes "not assigned". We write the map α in the form $X_{i_1} \mapsto a_{i_1}, \dots, X_{i_j} \mapsto a_{i_j}$ for $a_{i_1}, \dots, a_{i_j} \in \{0, 1\}$ with $1 \leq i_1 < i_2 < \cdots < i_j \leq n$, i.e. we omit the unassigned variables.

(b) In the setting of (a), if α is given by $X_1 \mapsto a_1, \dots, X_n \mapsto a_n$ for $a_1, \dots, a_n \in \{0, 1\}$, we call α a **complete assignment** of the logical variables $X_1, \dots, X_n$. In this case, we write also α as a tuple $a = (a_1, \dots, a_n) \in \{0, 1\}^n$.

(c) An assignment of the logical variables $X_1, \dots, X_n$ of the form $X_{i_1} \mapsto a_{i_1}, \dots, X_{i_j} \mapsto a_{i_j}$ for $a_{i_1}, \dots, a_{i_j} \in \{0,1\}$ with $1 \le i_1 < i_2 < \cdots < i_j \le n$ is called a **partial assignment** of the logical variables $X_1, \dots, X_n$ if there exists $k \in \{1, \dots, n\}$ such that $k \notin \{i_1, \dots, i_j\}$.

(d) Given $a \in \{0,1\}^n$, $\varphi(a)$ denotes the truth value obtained by assigning the variables X_i in the formula φ to a_i for $i = 1, \dots, n$.

(e) Given a partial assignment α given by $X_{i_1} \mapsto a_{i_1}, \dots, X_{i_j} \mapsto a_{i_j}$, $\varphi|_\alpha$ denotes the Boolean formula obtained by assigning the variables X_{i_j} in the formula φ to a_{i_j} for all $a_{i_1}, \dots, a_{i_j} \in \{0,1\}$.

(f) The complete assignment $a \in \{0,1\}^n$ is called a **satisfying assignment** or a **model** for φ if $\varphi(a) = 1$.

(g) The **set of satisfying assignments** (or **models**) for a propositional formula φ in n logical variables is defined as

$$\mathcal{S}(\varphi) = \big\{a \in \{0,1\}^n \mid \varphi(a) = 1\big\}.$$

(h) We say that $\varphi_1, \dots, \varphi_m$ **semantically imply** ψ if $\mathcal{S}(\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_m) \subseteq \mathcal{S}(\psi)$. In this case we write $\varphi_1, \dots, \varphi_m \models \psi$.

To the previous definition we associate the following *decision* problem.

Problem:	The Boolean satisfiability problem (SAT)
Input:	A set of clauses C in n logical variables.
Question:	Is there a model $a \in \mathbb{F}_2^n$ such that $a \in \mathcal{S}(C)$?

The problem SAT is known to be NP-complete. In other words, SAT is in NP and reducible to PSS in polynomial time. The *search* version of the problem is to find a satisfying assignment if one exists.

The following proposition establishes a connection between Boolean formulas in CNF and Boolean functions.

Proposition 2.18. *(a) Given a Boolean formula φ in n variables, there exists a Boolean function ψ in n variables such that $\varphi(a) = \psi(a)$ for all $a \in \mathbb{F}_2^n$.*

(b) Every propositional logic formula φ in n variables can be encoded in a Boolean formula ψ in n variables which is in CNF and satisfies $\psi(a) = \varphi(a)$ for all $a \in \mathbb{F}_2^n$.

Proof. (a) The function ψ is given by $a \mapsto \varphi(a)$.

(b) We show that

$$\varphi \equiv \bigwedge_{(a_1,\dots,a_n)\in F} \Bigg(\bigvee_{i \,|\, a_i = 0} X_i \bigvee_{j \,|\, a_j = 1} \bar{X}_j \Bigg) = \psi$$

for $F = \{a \in \mathbb{F}_2^n \mid \varphi(a) = 0\}$. Indeed, we have $a \in F$ if and only if $\psi(a) = 0$, which concludes the proof. □

From the logic point of view, a formula in ANF is constructed from the logical `XOR` operation and the logical `AND` operation. By Proposition 2.18 and 2.12, we can consider Boolean polynomials, Boolean functions, and propositional logic formulas (in CNF) as equivalent from the semantic point of view – they are just different representations of the same concept.

Example 2.19. The Boolean polynomial $f = x_1x_2 + x_1x_3 + 1 \in \mathbb{B}_3$ corresponds to the formula $\psi = (X_1 \wedge X_2) \oplus (X_1 \wedge X_3) \oplus 1$. In other words, we have $f(a) = \psi(a)$ for all $a \in \{0,1\}^3$. Note that the corresponding CNF formula is equal to $\psi = (\bar{X}_1 \vee \bar{X}_2 \vee X_3) \wedge (\bar{X}_1 \vee X_2 \vee \bar{X}_3)$. △

Rules of inference are used to derive new formulas from premises.

Definition 2.20. Let $\varphi_1, \ldots, \varphi_m, \psi$ be propositional formulae in n logical variables.

(a) A **rule of inference** ri (of propositional logic) transforms **premises**, i.e. a set of formulae, into a **conclusion**, i.e. a formula. Rules of inference are given in the following form

$$\frac{\varphi_1 \quad \varphi_2 \quad \cdots \quad \varphi_m}{\psi} \quad (\mathsf{ri})$$

where φ_i are the premises and ψ is the conclusion. To simplify the notation, we also write $\varphi_1, \ldots, \varphi_m \vdash_{\mathsf{ri}} \psi$.

(b) Let ri be a rule of inference. We say that the rule of inference ri is **correct** (or **sound**) if $\varphi_1, \ldots, \varphi_m \vdash_{\mathsf{ri}} \psi$ implies $\varphi_1, \ldots, \varphi_m \models \psi$.

(c) Let ri be a rule of inference. We say that the rule of inference ri is **complete** if $\varphi_1, \ldots, \varphi_m \models \psi$ implies $\varphi_1, \ldots, \varphi_m \vdash_{\mathsf{ri}} \psi$.

It is meaningful to require that rules of inference are always sound, i.e. that they preserve truth.

2.3 Algebraic Solvers

Solvers that search for more suitable generators of a given Boolean ideal are referred to as *algebraic solvers*. Thus, they can be used to solve the problem PSS. Note that they actually solve "harder" problems than PSS because they describe the whole solution set. The basic principle of algebraic solvers is to *generate* new polynomials in the ideal and to *simplify* the newly derived polynomials by the old ones in order to find new leading terms. For an overview of algebraic solving techniques, see [17, Ch. 12]

Through the section we focus on algebraic solvers working over Boolean polynomials. Thus, we set $P = \mathbb{F}_2[x_1, \ldots, x_n]$ and $F = \langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$.

Algebraic solvers work with polynomials in ANF and use polynomial calculus (see [89]) as the main inference rule, i.e.,

$$\frac{f \quad g}{f+g} \quad \text{(pca)} \qquad\qquad \frac{f}{x_i f} \quad \text{(pcm)}$$

where $f, g \in \mathbb{B}_n$, and x_i is an indeterminate. Polynomial calculus is sound (every $\mathbb{F}_2$-rational zero of the premises is an $\mathbb{F}_2$-rational zero of the derived polynomial) and complete (every semantic consequence, i.e. a polynomial in the ideal generated by the premises, can be derived by a sequence of inference rules).

To start with, we present the concept of Gröbner bases (GB), the Gröbner basis algorithm (GBA) and their Boolean variants.

Definition 2.21. Let I be an ideal in $P = K[x_1, \ldots, x_n]$ with a field K. Let σ be a term ordering. Let $G = \{g_1, \ldots, g_s\} \in I \setminus \{0\}$ and $F = \langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$.

(a) The set of polynomials G is called a **σ-Gröbner basis** of I if $\mathrm{LT}_\sigma(I) = \langle \mathrm{LT}_\sigma(g_1), \ldots, \mathrm{LT}_\sigma(g_s) \rangle$.

(b) We say that G is a **reduced σ-Gröbner basis** of I if the following conditions are satisfied.

   (i) For $i = 1, \ldots, s$, the coefficient of the term $\mathrm{LT}_\sigma(g_i)$ in g_i is equal to 1.

   (ii) The set $\{\mathrm{LT}_\sigma(g_1), \ldots, \mathrm{LT}_\sigma(g_s)\}$ is a minimal set of generators of $\mathrm{LT}_\sigma(I)$.

   (iii) For $i = 1, \ldots, s$, we have $\mathrm{Supp}(g_i - \mathrm{LT}_\sigma(g_i)) \cap \mathrm{LT}_\sigma(I) = \emptyset$.

(c) Let $K = \mathbb{F}_2$, $F \subset I$, and let $G \subset P$ be a finite set of Boolean polynomials in ANF. The set G is called a **Boolean σ-Gröbner basis** (or a **reduced Boolean σ-Gröbner basis**) of I if there exists a set $F' \subseteq \{x_1^2 + x_1, \ldots, x_n^2 + x_n\}$ such that $G \cup F'$ is a σ-Gröbner basis (or a reduced σ-Gröbner basis) of I.

The last definition follows from the discussion in [22, Sect. 2.2] and Proposition 2.12. Note that reduced Gröbner bases are unique. A standard way how to obtain a Gröbner basis is given by the famous *Buchberger algorithm* introduced in his PhD thesis [25]. New polynomials in Buchberger's algorithm are constructed by forming S-polynomials.

Definition 2.22. Let $f_i, f_j \in P$. The polynomial of form $S(f_i, f_j) = t_{ij} f_i - t_{ji} f_j$ where $t_{ij} = \mathrm{lcm}(\mathrm{LT}_\sigma(f_i), \mathrm{LT}_\sigma(f_j)) / \mathrm{LT}_\sigma(f_i)$ is called the **S-polynomial** of f_i and f_j.

The simplification is done by computing *normal remainders* (see [72, Sect. 1.6]), i.e. by the division algorithm with remainder. The algorithm is referred to as `NR`. The output of $\mathtt{NR}(s, S)$ for $s \in P$ and $S \subset P$ is the remainder of division s by S. (The algorithm `NR` requires S to be a tuple of polynomials. Thus the set S is viewed as a tuple of its elements.) Simple modifications of Buchberger's algorithm gives us the Boolean Buchberger algorithm described in Algorithm 2.1.

Algorithm 2.1 and many of its variants have several drawbacks. E.g., they tend to create high degree S-polynomials (however, this problem is more striking for non-Boolean versions). Subsequently, these high-degree polynomials are rewritten using `NR`, which may be a very time-consuming procedure. In practice, the algorithm spends the most of its time in reducing the last S-polynomial.

Algorithm 2.1 can be improved as follows.

Algorithm 2.1 `Buchberger` (The Boolean Buchberger Algorithm)

Input: A set of polynomials $S = \{f_1, \ldots, f_s\} \subset P = \mathbb{F}_2[x_1, \ldots, x_n]$ in ANF and a term ordering σ.

Output: A Boolean σ-Gröbner basis of an ideal $I = \langle S \rangle$ such that $F \subset I \subset P$, and F is the field ideal.

Require: The algorithm `NR` for computing normal remainders in [72, Th. 1.6.4].

1: $S' := S$
2: Let Q be the set of all pairs (f_i, f_j) such that $1 \leq i < j \leq s$.
3: **while** $Q \neq \emptyset$ **do**
4: Select $(f, g) \in Q$ and remove it from Q.
5: Let s be the S-polynomial of f and g.
6: $r := \texttt{NR}(s, (F \cup S'))$
7: **if** $r \neq 0$ **then**
8: $Q := Q \cup \{(r, h) \mid h \in S'\}$
9: $S' := S' \cup \{r\}$
10: **end if**
11: **end while**
12: **return** S'

- *Pair selection.* A careful selection of the pair in (f, g) in Step 3 greatly affects the running time of the algorithm. The ultimate goal is to find simple polynomials (i.e. with low degree and not many terms in the support) as soon as possible because they simplify the system the most.
- *Predicting zero reductions.* During a run of the algorithm, many S-polynomials are rewritten to zero, i.e. they are useless because they do not provide a new leading term. There exist criteria that allow the algorithm to skip some of those S-polynomials (see [93]).
- *Reducing many S-polynomials at once.* The polynomials in S can be represented as the coefficient matrix $M = \mathrm{CM}_\sigma(S)$. Firstly, we add new rows to M that correspond to S-polynomials of polynomials in S. Secondly, we append rows that correspond to possible reducers, i.e. polynomials of shape $t \cdot f$ with $f \in S$ and $t \in \mathbb{S}^n$ to the matrix M. Finally, computing REF of the whole matrix results in reducing all S-polynomials simultaneously (see [42]).

Algorithm 2.1 does not implement any sophisticated reasoning in the ring of Boolean polynomials. Rewriting using the field ideal can be done on a “deeper” level and not “outside” in `NR`. For instance, we point out the implementation of Boolean Gröbner basis algorithm using zero-suppressed decision diagrams in [22, Ch. 4]. For the (Boolean) Gröbner basis algorithm, the library `PolyBoRi` [17] and the `FGb` library [42] provide efficient actual implementations of such solvers. For further details of the theory of Boolean Gröbner bases, we refer to [22, Ch. 2].

Linear algebra methods emerge naturally in polynomial system solving. One very

common technique is *linearization* of a polynomial system. For instance, one can linearize the system by introducing auxiliary indeterminates for each non-linear term in the support and solve the resulting linear system by Gaußian elimination. This idea has evolved into methods such as XL and XSL [33]. Note that this technique is quite similar to the last improvement of Algorithm 2.1. We state the basic idea of linearization as Algorithm 2.2.

Algorithm 2.2 `Linearize` (The Linearization Algorithm)

Input: A set $S = \{f_1, \ldots, f_s\} \subset \mathbb{B}_n$ in ANF with $\mathcal{Z}(S) \neq \emptyset$, a term ordering σ, a degree bound $d \in \mathbb{N}_+$.
Output: A zero $a' \in \mathcal{Z}(S)$; the string `UNKNOWN` otherwise.

1: $Q := S \cup \{t \cdot f \mid t \in \mathbb{S}^n,\ f \in S,\ \deg(t \cdot f) \leq d\}$
2: $M := \mathrm{CM}_\sigma(Q)$
3: Let R be the solution space of the linear system $Mx = 0$ over $\mathbb{F}_2$.
4: **for** $a \in R$ **do**
5: Using the assignment a for terms in $\mathrm{Supp}(Q)$, deduce the assignment a' for the indeterminates $x_1, \ldots, x_n$.
6: **if** $f_i(a') = 0$ for all $i = 1, \ldots, s$ **then**
7: **return** a'
8: **end if**
9: **end for**
10: **return** `UNKNOWN`

The success of Algorithm 2.2 depends on a degree bound d. Note that each solution in the set R, which is a finite (but potentially large) set, has to be verified. The solution of the linearized system does not have to be compatible with the relations induced by terms as illustrated in the next example.

Example 2.23. Let $f_1 = x_1x_2 + x_2x_3 \in \mathbb{B}_3$ and $f_2 = x_1 \in \mathbb{B}_3$. The linearization of f_1 yields $g_1 = y_1 + y_2$ with the new indeterminates y_1, y_2. The polynomial f_2 is already linear. The assignment $(y_1, y_2, x_1) = (1, 1, 0)$ satisfies the linear system $g_1 = f_2 = 0$. However, the assignment does not induce a valid solution for the indeterminates x_1, x_2, x_3 of the initial system $f_1 = f_2 = 0$, because $1 = y_1 = x_1x_2$ is equal to zero under the assignment $x_1 = 0$. △

The linear system $Mx = 0$ in Algorithm 2.2 can be solved by Gaußian elimination, e.g. by computing a row echelon form of M. There are many improvements of this approach. For instance, if the elimination yields a univariate polynomial h, then the solution of the equation $h = 0$ can be used to branch the computation for the different cases in $\mathcal{Z}(h)$, or a newly derived polynomial of small degree can be inserted into S and treated as an input polynomial (see the concept of *mutants* in [2]).

Another approach to algebraic solving is *elimination of linear polynomials* such as performed by the algorithm `ElimLin` [34]. Its main idea is to perform LT-interreduction in order to obtain new linear equations. These linear equations are then substituted into

the system. The procedure goes on eliminating indeterminates in the system until there is no linear equation left. The idea of `ElimLin` is outlined in Algorithm 2.3.

Algorithm 2.3 `ElimLin` (The ElimLin Framework)

Input: A set of polynomials $S = \{f_1, \dots, f_s\} \subset \mathbb{B}_n$ in ANF and a term ordering σ.
Output: A set of linear polynomials L and a simplified set of polynomials $S' \subset \mathbb{B}_n$ such that $\langle L \cup S' \rangle = \langle S \rangle$.

1: $L := \emptyset$
2: $S' := S$
3: $Q := \emptyset$
4: **repeat**
5:   **if** $1 \in Q$ **then**
6:     **return** $(\{1\}, \emptyset)$
7:   **else**
8:     **for** $\ell \in Q$ **do**
9:       $L := L \cup \{\ell\}$
10:       $S' := S' \setminus \{\ell\}$
11:       Rewrite S' using $\mathrm{LT}_\sigma(\ell) \mapsto \ell - \mathrm{LT}_\sigma(\ell)$.
12:     **end for**
13:   **end if**
14:   LT_σ-interreduce S' and save the result to S'.
15:   Let Q be the set of all linear polynomials in S'.
16: **until** $Q = \emptyset$
17: **return** (L, S')

Algorithm 2.3 is guided by the term ordering σ. In the version of [34], there is certain freedom of choice which indeterminates are substituted and how the interreduction is performed. Even though the algorithm seems to be easy to implement, it is actually very difficult to handle the substitutions efficiently.

2.4 SAT Solvers

Algorithms which search for a satisfying assignment for a set of clauses are called *SAT solvers*. In this section we focus mainly on DPLL-based solvers. However, there exist other complete solvers such as lookahead solvers, or hybrid approaches like cube-and-conquer (e.g., see [52]). In order to find the whole solution space or count the number of solutions, so-called enumeration or #SAT solvers are used. In this section we assume that a set of clauses C is always finite. The section is based on [17].

Resolution is probably the most used inference rule in SAT solvers.

Definition 2.24. Let C be a (finite) set of clauses. Let $c, c' \in C$ be clauses, and let X_i be a logical variable.

(a) **Resolution** corresponds to the rule of inference

$$\frac{c \cup \{X_i\} \qquad c' \cup \{\bar{X}_i\}}{c \cup c'} \quad \text{(res)}$$

where neither X_i nor $\bar{X}_i$ are contained in $c \cup c'$.

(b) The conclusion $c \cup c'$ of the resolution rule in (a) is called the **resolvent** of $c \cup \{X_i\}$ and $c' \cup \{\bar{X}_i\}$.

(c) **Unit propagation** (UP, `up`), sometimes called **unit resolution**, is the special case of the resolution rule defined in (a) in which either c or c' is the empty clause.

(d) **Boolean constraint propagation** (BCP) refers to the simplification of C described by the following steps.

  (i) Let $U = \emptyset$.

  (ii) While there exists a unit clause $c = \{L\} \in C$ with a literal L, and while C does not contain the empty clause, repeat Steps (iii)–(v).

  (iii) Remove c from C, and append c to U.

  (iv) Remove all clauses from C which contain the literal L.

  (v) Remove the literal $\bar{L}$ from every clause in C which contains the literal $\bar{L}$.

  (vi) Return $C \cup U$.

(e) Resolution closures are defined inductively as follows.

  (i) $\mathrm{Res}^0(C) = C$

  (ii) $\mathrm{Res}^1(C) = C \cup \{r \mid r \text{ is resolvent of two clauses in } C\}$

  (iii) $\mathrm{Res}^{i+1}(C) = \mathrm{Res}(\mathrm{Res}^i(C))$ for $i > 1, i \in \mathbb{N}$.

  (iv) The set $\mathrm{Res}^\infty(C) = \bigcup_{i=0}^\infty \mathrm{Res}(C)$ is called the **resolution closure** of C.

Resolution is *sound* (i.e., the resolvent evaluates to `True`, when the original clauses both evaluate to `True`) and *not complete* (e.g., $c' = \{X_1, X_2\}$ is `True` if $c = \{X_1\}$ is `True`, but c' cannot be derived by resolution). Note that BCP corresponds to an iterative application of unit propagation.

Note that $\mathrm{Res}^i(C) \subseteq \mathrm{Res}^{i+1}(C)$ for $i \in \mathbb{N}$. Since there are only finitely many clauses using the variables in $\mathrm{Var}(C)$, there exists an index $j \in \mathbb{N}$ such that $\mathrm{Res}^j(C) = \mathrm{Res}^{j+1}(C)$, i.e. $\mathrm{Res}^\infty(C) = \mathrm{Res}^j(C)$.

The next proposition is well-known and tell us that resolvents can be added to the initial formula without modifying the set of models (see Claim (a)) and that the resolution closure can be used to determine the (un)satisfiability of a CNF formula (see Claim (b)). The proof can be found in [27, Sect. 4.1].

Proposition 2.25. *Let C be a finite set of clauses.*

(a) Let r be the resolvent of two clauses $c_1, c_2 \in C$. Then C is equivalent to $C \cup \{r\}$. In particular, we have $\mathcal{S}(C) = \mathcal{S}(C \cup \{r\})$.

(b) We have $\mathcal{S}(C) = \emptyset$ if only if $\emptyset \in \mathrm{Res}^\infty(C)$.

Because BCP takes almost all the running time of a SAT solver, it is a very important routine of SAT solvers. Its main task is to find new unit clauses and to detect a contradiction. We state a procedure for BCP under a given partial assignment in Algorithm 2.4. In the algorithm, we use $C|_{D \cup D'}$ for the Boolean formula obtained by the substitution of two distinct (i.e. containing different logical variables) partial assignments D and D' into a set of clauses C.

Algorithm 2.4 `BCP` (Boolean Constraint Propagation)

Input: A set of clauses C in the logical variables $X_1, \ldots, X_n$, a partial assignment D of the logical variables $X_1, \ldots, X_n$.

Output: A partial assignment D' of the logical variables $X_1, \ldots, X_n$ such that $C|_D \models C|_{D'}$ and a Boolean value a. If $a =$ `False`, then $\mathcal{S}(C|_D) = \emptyset$.

1: Let D' be the empty assignment.
2: **while** $\{L\}$ is a unit clause in $C|_{D \cup D'}$ for some literal L **do**
3:   **if** L is positive **then**
4:     Append the assignment $\mathrm{Var}(L) \mapsto 1$ to D'.
5:   **else**
6:     Append the assignment $\mathrm{Var}(L) \mapsto 0$ to D'.
7:   **end if**
8: **end while**
9: **if** $C|_{D \cup D'}$ contains the empty clause **then**
10:   **return** $(D', \texttt{False})$
11: **else**
12:   **return** $(D', \texttt{True})$
13: **end if**

An example of a run of Algorithm 2.4 is given below.

Example 2.26. Let $C = \{c_1, c_2, c_3\}$ be the set of clauses with $c_1 = \{X_1, X_4\}$, $c_2 = \{\bar{X}_1, X_2, X_4\}$ and $c_3 = \{X_1, X_3, X_4\}$. Consider the partial assignment D given by$X_4 \mapsto 0$. Algorithm 2.4 detects that the clause $c_1 = \{X_1, X_4\}$ becomes a unit clause in $C|_D$, i.e. we initialize D' by the assignment $X_1 \mapsto 1$. It continues with the clause $c_2 = \{\bar{X}_1, X_2, X_4\}$ and adds $X_2 \mapsto 1$ to D'. Note that the clause $c_3 = \{X_1, X_3, X_4\}$ is satisfied by $D \cup D'$, and hence it is not considered. Thus the algorithm returns the partial assignment D' given by $X_1 \mapsto 1, X_2 \mapsto 1$. △

Note that Algorithm 2.4 can be rewritten such that the set of clauses C is being modified during the assignments into C. However, we prefer the version given above because it is closer to the real implementation of SAT solvers, e.g. the input clauses are never modified, etc.

Next we present the famous *Davis-Putnam-Logemann-Loveland* (DPLL) Algorithm and its subroutines. They help us to understand the modern data structures used in SAT solvers. The DPLL algorithm is a depth-first search procedure. The DPLL algorithm makes *decisions*, i.e. it chooses an assignment for a variable and propagates it using BCP.

The order of decisions is very important, and it is stored in special data structures called decision stacks. (For the definition of an assignment and underlying notations, see Definition 2.17.)

Definition 2.27. (a) A **decision stack** D is defined as a two-dimensional tuple of assignments of logical variables, i.e. $D = (D_0, \dots, D_s)$ where D_i is a tuple which contains assignments of logical variables for $i = 1, \dots, s$. We write the tuple D_i as $D_i = (X_{i_1} \mapsto a_{i_1}, \dots, X_{i_j} \mapsto a_{i_j})$ with $a_{i_1}, \dots, a_{i_j} \in \{0, 1\}$.

(b) In the setting of (a), the position of the tuple D_i in D (i.e. the index i) is called the **decision level** of D_i.

(c) Let C be a set of clauses, and let $D = (D_0, \dots, D_s)$ be a decision stack. The set of clauses $C|_D$ denotes the set of clauses $C|_\alpha$ where the partial assignment α is defined by all assignments of the variables in $\bigcup_{i=0}^{s} D_i$.

Notice that the indexing of the tuples in D starts from 0. As usual for stacks, the two main operations for the decision stacks are insertions of a new tuple at the top of the stack and removals of the latest tuple inserted into the stack.

Before we introduce a framework for DPLL algorithms, we describe its very important subroutine for *chronological backtracking* (e.g., see [17, Sect. 3.6.1]) in Algorithm 2.5. The idea of the algorithm is to backtrack to the last branch of the decision tree from which the search space has not been explored for both values of a logical variable.

The set A in Algorithm 2.5 contains a history of some decisions which were made in the main DPLL routine such that we can keep track of which variables were toggled. The Boolean value d is `False` if and only if the decision stack is empty.

Now we are ready to describe a framework for DPLL in Algorithm 2.6.

In Algorithm 2.6, an assignment of a variable in Step 10 is called a decision. The Boolean value b encodes the fact whether a decision has to be made in the next iteration of the DPLL procedure. The assignments derived using BCP (i.e. using Algorithm 2.4) are called *implied assignments*.

Note that Algorithm 2.6 does not perform a vanilla brute search, i.e. it does not search through all decisions with a subsequent verification if the current assignment satisfies the formula, but it uses logical consequences of BCP to prune the search space. This additional feature does not change the fact that the DPLL algorithm is finite and correct.

Let us illustrate the synergy of the subroutines of DPLL in the following examples. Example 2.28 is the modified Example 3.2 in [102].

Algorithm 2.5 `Chro-Backtrack` (Chronological Backtracking)

Input: A set A of assignments of the logical variables $X_1, \dots, X_n$. A decision stack D which consists of $s+1$ tuples.

Output: An updated decision stack D', a Boolean value d.

1: $D' := D$
2: Let $Y_i \mapsto a_i$ denote the first assignment of a variable in the tuple T where T is located at position i in D' for $i = 1, \dots, s$ where $Y_i \in \{X_1, \dots, X_n\}$.
3: **for** $k = s, \dots, 0$ **do**
4:   **if** $k = 0$ **then**
5:     $s' := 0$
6:     **break**
7:   **else if** $Y_k \mapsto \bar{a}_k$ is not contained in A **then**
8:     $s' := k$
9:     **break**
10:   **end if**
11: **end for**
12: **if** $s' = 0$ **then**
13:   **return** $(D', \texttt{False})$
14: **else**
15:   Remove the tuples at positions $s', \dots, s$ from D'.
16:   $A := A \setminus \{Y_{s'+1} \mapsto a_{s'+1}, \dots, Y_s \mapsto a_s, Y_{s'+1} \mapsto \bar{a}_{s'+1}, \dots, Y_s \mapsto \bar{a}_s\}$
17:   Append the tuple $(Y_{s'} \mapsto \bar{a}_{s'})$ to D'.
18:   $A := A \cup \{Y_{s'} \mapsto \bar{a}_{s'}\}$
19:   **return** $(D', \texttt{True})$
20: **end if**

Example 2.28. Consider the following clauses.

$$\begin{aligned}
c_1 &= \{\bar{X}_1, X_2\} \\
c_2 &= \{\bar{X}_1, X_3, X_9\} \\
c_3 &= \{\bar{X}_2, \bar{X}_3, X_4\} \\
c_4 &= \{\bar{X}_4, X_5, X_{10}\} \\
c_5 &= \{\bar{X}_4, X_6, X_{11}\} \\
c_6 &= \{\bar{X}_5, \bar{X}_6\} \\
c_7 &= \{X_1, X_7, \bar{X}_{12}\} \\
c_8 &= \{X_1, X_8\} \\
c_9 &= \{\bar{X}_7, \bar{X}_8, \bar{X}_{13}\} \\
c_{10} &= \{\bar{X}_{14}\} \\
c_{11} &= \{X_{14}, X_{15}\}
\end{aligned}$$

Assume that the decisions are as follows (in order of appearance) $X_9 \mapsto 0$, $X_{10} \mapsto 0$, $X_{11} \mapsto 0$, $X_{12} \mapsto 1$, $X_{13} \mapsto 1$, $X_1 \mapsto 1$. After each decision is made, we run BCP to derive implied assignments. The corresponding decision stack D is given below.

Algorithm 2.6 DPLL (The DPLL Algorithm)

Input: A set of clauses C in n logical variables $X_1, \ldots, X_n$.
Output: A decision stack which contains a satisfying assignment in $\mathcal{S}(C)$ if $\#\mathcal{S}(C) > 0$; `False` otherwise.
Require: Algorithms 2.4 and 2.5.

1: $(D', a) :=$ `BCP`$(C, \emptyset)$ where $\emptyset$ denotes the empty assignment
2: **if** $a =$ `False` **then**
3:   **return** `False`
4: **end if**
5: $D := (D')$
6: $A := \emptyset$
7: $b :=$ `True`
8: **while** D does not contain a complete assignment of the variables $X_1, \ldots, X_n$ **do**
9:   **if** $b =$ `True` **then**
10:     Choose a logical variable $X_i \in \mathrm{Var}(C)$ which is not assigned in D and a Boolean value $v \in \{0, 1\}$.
11:     Append the tuple $(X_i \mapsto v)$ to D
12:     $A := A \cup \{X_i \mapsto v\}$
13:   **end if**
14:   $b :=$ `True`
15:   $(D', a) :=$ `BCP`(C, D)
16:   Append the assignments in D' to the end of the last tuple in D.
17:   **if** $a =$ `False` **then**
18:     $(D, d) :=$ `Chro-Backtrack`(A, D).
19:     $b :=$ `False`
20:     **if** $d =$ `False` **then**
21:       **return** `False`
22:     **end if**
23:   **end if**
24: **end while**
25: **return** D

decision level	*decision*	*implied assignments*
0		$X_{14} \mapsto 0$, $X_{15} \mapsto 1$
1	$X_9 \mapsto 0$	
2	$X_{10} \mapsto 0$	
3	$X_{11} \mapsto 0$	
4	$X_{12} \mapsto 1$	
5	$X_{13} \mapsto 1$	
6	$X_1 \mapsto 1$	$X_2 \mapsto 1$, $X_3 \mapsto 1$, $X_4 \mapsto 1$, $X_5 \mapsto 1$, $X_6 \mapsto 1$, $X_6 \mapsto 0$

We can see that these decisions yield a contradiction at decision level 6. △

Let us continue with Example 2.28 and apply Algorithm 2.5 as described in Algorithm 2.6.

Example 2.29. In the setting of Example 2.28, we assume that $A = \{X_9 \mapsto 0,\, X_{10} \mapsto 0,\, X_{11} \mapsto 0,\, X_{12} \mapsto 1,\, X_{12} \mapsto 0,\, X_{13} \mapsto 1,\, X_{13} \mapsto 0,\, X_1 \mapsto 1,\, X_1 \mapsto 0\}$. Algorithm 2.5 backtracks to the following decision stack.

decision level	*decision*	*implied assignments*
0		$X_{14} \mapsto 0,\, X_{15} \mapsto 1$
1	$X_9 \mapsto 0$	
2	$X_{10} \mapsto 0$	
3	$X_{11} \mapsto 1$	

The set A is updated to $A = \{X_9 \mapsto 0,\, X_{10} \mapsto 0,\, X_{11} \mapsto 0,\, X_{11} \mapsto 1\}$. △

Most modern SAT solvers are based on a *conflict-driven conflict-learning* (CDCL) procedure, i.e. on DPLL with the addition of clause learning. A conflict means that a certain partial assignment yields a contradiction. In the following we mention only the basic ideas of CDCL which we will use in the later chapters. For a detailed background on CDCL solvers, see [17, Ch. 4]. The CDCL solvers generate new clauses, called *learned 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 in a CDCL way are `MiniSAT` [41] and `Glucose` [6].

Let us explain the main differences to the DPLL procedure given in Algorithm 2.6. We start with new subroutines used in CDCL, namely, a procedure for learning new clauses and for non-chronological backtracking. Let C be the input set of clauses.

- *Conflict analysis.* This function is called whenever a contradiction is discovered by BCP under the current assignment. During this stage, the conflict is analyzed, and a new set of so-called learned clauses C' with $C \models C'$ is appended to C. (Note that in practise the learned clauses are stored in a separate database and not directly appended to the input clauses.) The clauses in C' should encode the reason why the conflict has happened and prevent the same conflict conditions. Typically, it holds that $C \vdash_{\mathsf{res}} C'$, i.e. the learned clauses are derived using resolution.

- *Non-chronological backtracking.* This function replaces `Chro-Backtrack`. The function analyzes the learned clauses and computes the backtrack point, i.e. to which point in the run of the CDCL algorithm we return and which assignment of a variable is flipped. In contrast to chronological backtracking in Algorithm 2.5, we are able to backtrack to a higher level in the search tree.

The tight integration of the above functions is the key of the success of modern SAT solvers. Without going into the details, we illustrate the synergy of the two functions in an example using the most common strategy for learning based on resolution and *first unique implication points* (1UIP, see [17, Sect. 3.6.4.]).

Example 2.30. In the setting of Example 2.28, we discovered that the clause c_6 yields a contradiction at the decision level 6. We recall the decision stack at that point of time. But this time we write the clause that became a unit in parentheses.

decision level	*decision*	*implied assignments*
0		$X_{14} \mapsto 0\,(c_{10})$, $X_{15} \mapsto 1\,(c_{11})$
1	$X_9 \mapsto 0$	
2	$X_{10} \mapsto 0$	
3	$X_{11} \mapsto 0$	
4	$X_{12} \mapsto 1$	
5	$X_{13} \mapsto 1$	
6	$X_1 \mapsto 1$	$X_2 \mapsto 1\,(c_1)$, $X_3 \mapsto 1\,(c_2)$, $X_4 \mapsto 1\,(c_3)$, $X_5 \mapsto 1\,(c_4)$, $X_6 \mapsto 1\,(c_5)$, $X_6 \mapsto 0\,(c_6)$

The conflicting clause is c_6. The key observation is that the literals in c_6 are complementary to the assignments in the decision stack. Moreover, we kept track of the clauses that contains the complementary literals. This means we can resolve the literals until we get a clause only with decision literals (i.e. the literals that contain variables appearing in the decisions). It turns out (empirically) that we can terminate the resolution chain earlier, and we get a better quality clause.

The idea is to resolve c_6 sequentially until we reach the 1UIP, i.e. the resolvent contains only one variable in the assignments in the decision stack at the most current decision, i.e. in our case one variable in the decision stack at level 6. We give the decision levels of the variables for each literal after the symbol "@".

Let us start resolving on X_6. The resolvent of c_6 and c_5 is equal to $r_1 = \{\bar{X}_4@6,\ \bar{X}_5@6,\ X_{11}@3\}$, which is not a UIP. We go on with X_5. Resolving r_1 with c_4 yields the clause $r_2 = \{\bar{X}_4@6,\ X_{10}@2,\ X_{11}@3\}$, which is the 1UIP and thus the learned clause. The learned clause r_2 is then appended to the input clauses. The set of models remains the same because of Proposition 2.25.

Next we backtrack to the highest level (excluding the current level) appearing in the learned clause, i.e. to level 3. The idea behind the 1UIP strategy is that the assignment $X_4 \mapsto 0$ is immediately set by `BCP` after backtracking because of r_2. Now the decision stack looks as follows.

decision level	*decision*	*implied assignments*
0		$X_{14} \mapsto 0$, $X_{15} \mapsto 1$
1	$X_9 \mapsto 0$	
2	$X_{10} \mapsto 0$	
3	$X_{11} \mapsto 0$	$X_4 \mapsto 0$

△

Note that in the previous example we skipped the whole subtree in the search tree in contrast to the chronological backtracking in DPLL. Thus the finiteness and correctness are more difficult to prove and require a richer notation. The learned clause corresponds to a cut in the *implication graph* in which vertices are the assignments in the decision stack, and edges represent which clause caused its UP. For more details, see [102].

The effectiveness of modern CDCL SAT solvers is based on the following ingredients.

- *Preprocessing and inprocessing.* The goal of a preprocessing is to reduce the formula size and decide if the formula is trivially (un)satisfiable. For instance, BCP can be run before the first decision as in Algorithm 2.6. For another example of preprocessing, see [40]. To add more reasoning to search, various inprocessing techniques are used, e.g. see [52].

- *Restarts.* Random restarts may help the solver to jump into a different part of the search tree, and hence deep "useless" areas in the search tree can be avoided. Note that the SAT solver does not throw away the learned clauses.

- *Unlearning.* Too many learned clauses slow down BCP. Hence various clause deletion strategies are used to reduce the number of learned clauses. The input clauses are never removed.

- *Heuristics.* Branching heuristics, i.e. rules for deciding which assignments to choose, influence the running time of a SAT solver a lot. Therefore they are a very important ingredient of CDCL solvers. For instance, *variable state independent decaying sum* (VSIDS) is such heuristics. It turns out that it is very important that the branching heuristic is cheap. For further details, see [17, Ch. 7] or [16].

- *Efficient data structures.* In a run of CDCL it is very important to identify contradictions and unit clauses. The evaluation does not have to be done by assigning all literals in a clause c whenever the SAT solver assigns variables in $\mathrm{Var}(c)$. In contrast, we may apply *lazy data structures*, such as *watched literals*, which help to identify conflicting and unit clauses in a very effective way. For further details, see [17, Sect. 4.5.1].

2.5 SMT and Programmatic SAT

Real-world logic problems are often expressed in some "richer" logic theory rather than propositional logic. The problem of deciding the satisfiability of propositional formulas with respect to some background theory is called *satisfiability modulo theory* (SMT). A formal introduction to SMT requires a heavy portion of notations and definitions that we will not need (see [102] or [17, Ch. 26]). Instead, we mention the basic principles of SMT without going into details.

For the whole section we fix some decidable (i.e. there is an effective method for determining membership of formulas in the theory) formal theory T, and we always assume that a formula in (the signature of) the theory T (so-called *T-formula*) is given in CNF, i.e. as a conjunction of disjunctions of atoms. Moreover, we restrict our attention to the satisfiability of quantifier-free formulae.

In the following examples we give formulas in the *theory of linear arithmetics on the integers* (TLAI) and in the *theory of fixed-width bit vectors* (TBV). For further details, we refer to [102, Sect. 4.2].

Example 2.31. (a) The formula $\varphi = (X_1 < X_2) \wedge \big((X_1 \le 0) \vee (X_2 < X_3)\big) \wedge \big((X_1 < X_3) \vee (X_3 \le 0)\big)$ is a TLAI-formula in CNF.

(b) Let $n = 32$. Consider the standard bitwise `XOR` and `AND` operations on the bit vectors of n bits. The formula $\varphi = (X_1 \oplus X_2 \oplus X_3 = X_4) \wedge (X_1 \wedge X_3 = X_2)$ is a TBV-formula in CNF. △

Boolean abstraction of a T-formula φ in CNF is a Boolean formula in CNF where the atoms of φ are substituted by new Boolean variables.

Example 2.32. In the setting of Example 2.31(a), the Boolean abstraction of the TLAI-formula φ is equal to $Y_1 \wedge (Y_2 \vee Y_3) \wedge (Y_4 \vee Y_5)$, where $Y_1, \ldots, Y_5$ are new Boolean variables. △

There are two approaches how to decide satisfiability of T-formulae.

- *Eager approach.* A T-formula is converted into an equisatisfiable CNF, i.e. the models of the two formulas are preserved, and a SAT solver is invoked. In the setting of TBV and hardware circuits, this approach is called *bit blasting*. On one hand, no special solver has to be tailored for the theory T. On the other hand, such direct CNF encodings can be quite inefficient for SAT solving and not very "readable". For further details, see [17, Sect. 26.3].

- *Lazy approach.* The initial T-formula is abstracted to a Boolean CNF formula. The Boolean abstraction is then fed to a SAT solver, which cooperates with a theory solver for T (so-called T-solver). The T-solver examines the model of the Boolean abstraction that was obtained from the SAT solver and tries to deduce a model for the T-formula. If the model of the Boolean abstraction cannot be extended to a model of the T-formula, the T-solver can provide the SAT solver a reason for this inconsistent assignment. For further details, see [17, Sect. 26.4].

The lazy approach is often implemented as DPLL(T), where the theory solver is used in a DPLL-based algorithm. A simple framework for DPLL(T) is presented in Algorithm 2.7. The algorithm is a simplified version of [17, Fig. 26.3]. Note that `DPLL` is used as an enumerator for models of C.

Notice that a DPLL solver is "blind" and cannot see the meaning of the literals appearing in the abstraction. In more sophisticated implementation of DPLL(T) the theory solver may be integrated tighter with the SAT solver than in Algorithm 2.7, e.g. the theory solver can help the SAT solver to choose the branching variables, it provides reasons why no models are found in Step 3, etc. We give an example of a run of Algorithm 2.7 where we illustrate the simple blocking mechanism.

Example 2.33. In the setting of Example 2.31(a) and Example 2.32, we have the TLAI-formula $\varphi = (X_1 < X_2) \wedge (X_1 \leq 0 \vee X_2 < X_3) \wedge (X_1 < X_3 \vee X_3 \leq 0)$ and its Boolean abstraction $\psi = Y_1 \wedge (Y_2 \vee Y_3) \wedge (Y_4 \vee Y_5)$. Consider the model $a = (1, 0, 1, 0, 1) \in \mathcal{S}(\psi)$. Note that the assignment a is not compatible with a valid assignment of atoms in φ because $X_1 < X_2$ and $X_2 < X_3$ imply $X_1 < X_3$. Thus we add the blocking clause $c = \{\bar{Y}_1, Y_2, \bar{Y}_3, Y_4, \bar{Y}_5\}$. △

Algorithm 2.7 `DPLL-T` (Framework for Lazy SMT Solving)

Input: A T-formula φ in CNF.
Output: A model for φ if one exists, `UNSAT` otherwise.
Require: A T-solver `TSolver`, a DPLL-based solver `DPLL` such as Algorithm 2.6.

1: Let C be a set of clauses corresponding to the Boolean abstraction of φ.
2: **while** `DPLL(`C`)` returns a model a **do**
3:    **if** there exists a model A for φ derived from a using `TSolver` **then**
4:       **return** A
5:    **else**
6:       Let c be a clause that blocks the assignment a.
7:       $C := C \cup \{c\}$
8:    **end if**
9: **end while**
10: **return** `UNSAT`

Addition of blocking clauses such as in Example 2.33 alone is not very efficient. We can do the learning phase in `TSolver` in a more sophisticated way and learn the true core for the conflict. The shorter the new clause is, the better the SAT solver performs.

Example 2.34. In Example 2.33, the clause c relates 5 variables $Y_1, \ldots, Y_5$, but the conflicting pieces were only Y_1, Y_3, Y_4. Moreover, the model $a' = (1, 1, 1, 0, 1)$ leads to the same conflict, and it is not blocked by adding c. We can encode the true core of the conflict, i.e. that the implications $X_1 < X_2$ and $X_2 < X_3$ imply $X_1 < X_3$, into the new clause $c' = \{\bar{Y}_1, \bar{Y}_3, Y_4\}$. △

One of the biggest drawbacks of lazy SMT solving is the necessity of implementing theory solvers for various logical theories. In practise, it is sometimes sufficient to integrate only "small pieces" of a theory into SAT solvers. One solution for such quick enhancements is called programmatic SAT solving.

We call a CDCL SAT solver *programmatic* if it is augmented with the *callback functions* that allow the user to add extra functionality to its propagation and conflict analysis routines. The callback function is called after each decision of the SAT solver. Programmatic SAT solving was introduced in [44], and it is in fact a variation of DPLL(T), which differs from the DPLL(T) concept in the following ways.

- The theory solver in the context of programmatic SAT can be an arbitrary piece of code written in a programming language such as C++ that implements the callback function. We remark that no special requirements on the code are required.
- The callback function is particularized to every input of the solver. That is, unlike the T-solver in DPLL(T) which remains invariant for all formulas from the language of T, the callback functions added via the programmatic interface are specific and unique to each input.

- The interface of the programmatic SAT solvers is much simpler than that of the SMT solvers. Hence, programmatic solvers are more flexible and tend to be more user-friendly.

Programmatic SAT solving allows easy customization of the SAT solver to specific Boolean problems. The SAT developer thus has more control over the power of a SAT solver. This architecture has also been found useful for solving various problems coming from combinatorics (see [23] or [24]).

2.6 An Overview of Other Solving Techniques

In this section we mention other methods than can be applied to solve the problem PSS or SAT (after a suitable conversion). Even though the section does not contain a complete list by far, it provides some useful references to other research areas. We do not write down all basic definitions that are required for understanding the new methods. The only exception is the following definition from the theory of CSP (*constraint satisfaction problems*) because it makes the notion of a "constraint" precise.

Definition 2.35. (a) A **CSP problem** is defined as a triple (X, D, C), where

- X is a set of variables, i.e. $X = \{x_1, \dots, x_n\}$.
- D is a set of domains of X, i.e. $D = \{D_{x_1}, \dots, D_{x_n}\}$ where the set D_{x_i} is the domain of the variable x_i.
- C is a set of constraints. Every constraint $c \in C$ is a tuple $c = (X_c, R_c)$ where $X_c = \{x_{i_1}, \dots, x_{i_s}\} \subset X$ is a set of variables, and R_c is a relation $R_c \subseteq D_{x_{i_1}} \times \cdots \times D_{x_{i_s}}$.

(b) In the setting of (a), let $c = (X_c, R_c) \in C$ be a constraint with $X_c = \{x_{i_1}, \dots, x_{i_s}\} \subseteq X$. We say that an assignment $x_{i_1} \mapsto a_{i_1}, \dots, x_{i_s} \mapsto a_{i_s}$ with $a_{i_j} \in D_{x_{i_j}}$ **satisfies** the constraint c if $(a_{i_1}, \dots, a_{i_s}) \in R_c$.

Some other solving methods that have not been mentioned (and will not be considered in this doctoral thesis) are given in the following list.

- *CSP solving.* Roughly speaking, SAT or SMT can be seen as a subset of CSP. Thus CSP solvers (i.e. the solver that solves CSPs) have a variety of applications. They implement various techniques for maintaining local consistency, backjumping, or constraint learning. For further details, see [98].

- *Brute-force search.* We can consider CDCL SAT solvers as tools for a "clever" brute-force search over CNF. There exists brute-force search algorithms over ANF with a clever evaluation mechanism such as the tool `libFES` [21]. Moreover, there exist algebraic techniques such as one in [81] that gain an exponential speedup over the vanilla brute-force approach, even though its idea is based on a brute-force search as well. However, the result in [81] seems to be interesting only for theoretical studies because of its asymptotic nature.

- *Combinatorial approaches.* The approach is based on the following observation. Given a set of sparse Boolean polynomials S in ANF, it is easy to compute $\mathcal{Z}(f)$ for $f \in S$. To get the set of common zeros $\mathcal{Z}(S)$, the individual sets of zeros $\mathcal{Z}(f)$ for $f \in S$ have to be "glued" together. Various gluing algorithms can be found in [103].

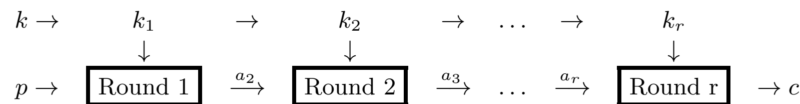
- *IP/MILP.* One can encode a problem in a set of linear inequalities. If the coefficients of the inequalities are real numbers, the solution can be found in polynomial time. However, in our setting the coefficients of the inequalities are typically defined over $\mathbb{Z}$ or 0/1. Such instances are dealt with in the theories of Integer Programming (IP), Mixed Integer Linear Programming (MILP), etc. (see [86]).

2.7 Block Ciphers and Hash Functions

In this section we look into how to encode various Boolean functions in ANF and CNF. In particular, we focus on cryptographic functions that are implemented in hardware (such as logic circuits or chips), and we provide a high-level description of iterated block ciphers and iterated hash functions. Other functions (e.g. non-cryptographic, software-based, etc.) can be encoded in a similar way. The section is not meant to be a complete description of cryptographic primitives with their security requirements. Instead, we rather analyze a few interesting cases. For more cryptographic background, we refer to [108].

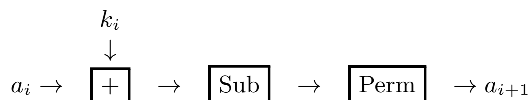
Hardware implementations have to satisfy a lot of conditions that are required by the customers and integrated circuit manufacturers. For instance, the complexity of an integrated circuit corresponds to the silicon area that is used, and thus corresponds to the cost of the circuit. That is why the complexity of the circuit is reduced by using simple transformations that are applied multiple times during the evaluation. In the next example, we give an overview of iterated block ciphers based on substitution–permutation networks, where the above reduction is clear.

Example 2.36. A *block cipher* is a Boolean map $E : \mathbb{F}_2^m \times \mathbb{F}_2^n \to \mathbb{F}_2^n$ mapping a m-bit key k and a n-bit plaintext p to a n-bit ciphertext c. For an *iterated cipher*, *subkeys* (also called *round keys*) $k_1, \ldots, k_r \in \mathbb{F}_2^n$ are derived from the main key k. This key generation phase is called the *key schedule* (also called the *key expansion*) of the cipher. The workflow of an iterated cipher is given below.



The round keys are applied in the round functions. A round function is a function $\mathbb{F}_2^n \times \mathbb{F}_2^n \to \mathbb{F}_2^n$ mapping the round key k_i and an intermediate state a_i to an updated state a_{i+1}. (We set $a_1 = p$ and $a_{r+1} = c$.)

A typical round function in a *substitution-permutation network* is given below. (The order of the operations may differ.)



The round function is a composition of key-dependent transformations such as bitwise XOR, a non-linear layer (substitution; very often realized by *S-Boxes*), and a linear layer (realized by a permutation). △

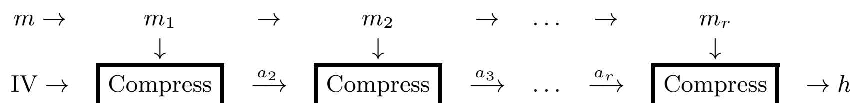
Of course, the resulting encryption has to be bijective for any choice of keys such that decryption is possible. Typical examples of the design outlined in Example 2.36 are AES [36], Serpent [18], and lightweight ciphers (i.e. with even smaller circuit complexity) LED [49] or PRESENT [19]. Different security aspects are considered when dealt with symmetric cryptosystems. The key is supposed to be private. The other parts such as the plaintext, the ciphertext, the key schedule and a complete specification of the cipher are public. (Sometimes the plaintext is unknown as well.) For instance, the key recovery attack is defined as follows.

Problem:	Key recovery (KEY)
Input:	An encryption function $E : A \times B \to B$, ciphertext-plaintext pairs $(p_1, c_1), \ldots, (p_k, c_k) \in B \times B$.
Question:	Find a key $k \in A$ with $E(k, p_i) = c_i$ for $i = 1, \ldots, k$.

The ciphertext-plaintext pairs in the problem KEY may be chosen by an attacker. In that case, we speak of a *chosen-plaintext attack*.

The concept of iteration can be also found in (unkeyed) hash functions.

Example 2.37. A *compression function* is a Boolean function $\mathbb{F}_2^{k+t} \to \mathbb{F}_2^k$ where $t \in \mathbb{N}_+$. The compression function is used to hash a binary string of arbitrary length in an iterative way. Consider a message $m \in \mathbb{F}_2^n$ for which we would like to compute the hash h. In the preprocessing, the message m is padded with additional bits using an injective *padding function* such that the length of the padded message m' is a multiple of t. Then m' is divided into bit string $m_1, \ldots, m_r$ of length t. The workflow of an *iterated hash function* is given below.



The compression function takes $m_i \in \mathbb{F}_2^t$ and $a_i \in \mathbb{F}_2^k$ as inputs and compresses the concatenation of m_i and a_i to a_{i+1}. (We have $a_1 = \mathrm{IV}$ with an *initialization vector* $\mathrm{IV} \in \mathbb{F}_2^k$ and $a_{r+1} = h$.) △

Typical hash functions that share their design with Example 2.37 are MD5 [97], SHA-1 or SHA-2 [39]. Security requirements of hash functions are specified by the difficulty of inverting the hash function or finding collisions. The specification of the compression function as well as the computed hash are public. The corresponding message may or may not be public. For instance, the preimage attack is defined as follows.

Problem:	Preimage (PREIMAGE)
Input:	A hash function $H : A \to B$ and its digest $h \in B$.
Question:	Find a message $m \in A$ with $H(m) = h$.

If the problem PREIMAGE cannot be solved efficiently for a hash function H, then H is said to be *preimage resistant*.

Now we focus on how to encode such iterative Boolean functions into CNF or ANF. However, the same procedure can be applied to any Boolean map such as one representing multiplier circuits (see [68]). In order to encode a Boolean function of the above types, we proceed as follows.

(1) Identify the basic functions (e.g. in round functions, in compresson functions, etc.).

(2) Find ANF (or CNF) encodings of the basic functions in (a).

(3) Introduce additional indeterminates (or logical variables) for intermediate states representing the input and output states of the basic functions.

(4) Connect the output variables of the preceding transformation with the input variables of the next transformation.

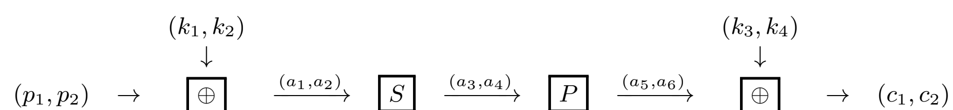
Step (1) is usually very easy and follows from the specification of the cryptographic algorithm. By Propositions 2.12 and 2.18, we know that ANF or CNF representations exist for any Boolean function (or for any Boolean map). However, the representation of the whole transformation is often not feasible without introducing new auxiliary variables in Step (3). Thus Steps (1)–(4) naturally reflect the workflow of the underlying hardware implementation.

There are two main approaches how to implement Step (2). Let φ be a Boolean map. Firstly, we can encode the equation in an *explicit* way as $y = \varphi(x)$ mapping the input variables x to the output variables y. Secondly, we may find an *implicit* function $\psi(x, y)$ such that the relation is satisfied if and only if $\varphi(x) = y$. Implicit ANF encodings usually have lower degrees than explicit ones.

Of course, the quality of the encodings matters. In the case of an ANF encoding, one can use the interpolation algorithm in [73, Thm. 6.3.10, Cor. 6.3.11], known as the *Buchberger-Möller algorithm*, that outputs already a Gröbner basis. In the case of a CNF encoding, *Tseitin transformations* or logic minimizers such as `Espresso` [99] may come in handy. There exist also hardware compilers that are able to convert a gate-level description into a CNF representation which use various minimization tricks which make it likely that the output CNF aids the SAT solvers.

To conclude this section, we encode a (not secure) toy cipher in ANF.

Example 2.38. Consider the encryption $E : \mathbb{F}_2^2 \times \mathbb{F}_2^2 \to \mathbb{F}_2^2$ given by the following diagram.



where $\oplus$ is component-wise XOR, and $S : \mathbb{F}_2^2 \to \mathbb{F}_2^2$ is the function defined by the following table.

(x_1, x_2)	$S(x_1, x_2)$
$(0, 0)$	$(0, 1)$
$(1, 0)$	$(0, 1)$
$(0, 1)$	$(1, 1)$
$(1, 1)$	$(0, 0)$

The function $P : \mathbb{F}_2^2 \to \mathbb{F}_2^2$ is defined by

$$P \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

The function S can be represented by two Boolean polynomials $s_1 = x_1x_2 + x_2$ and $s_2 = x_1x_2 + 1$ in $\mathbb{B}_2$, i.e. we have $S(x_1, x_2) = (x_1x_2 + x_2,\, x_1x_2 + 1)$. We also have $P(x_1, x_2) = (x_2,\, x_1 + x_2 + 1)$. Now we encode the transformation step by step. We start with the leftmost XOR and get Boolean polynomials f_1, f_2. Then we encode S in f_3, f_4.

$$\begin{aligned} f_1 &= a_1 + p_1 + k_1 \qquad & f_3 &= a_3 + a_1a_2 + a_2 \\ f_2 &= a_2 + p_2 + k_2 \qquad & f_4 &= a_4 + a_1a_2 + 1 \end{aligned}$$

Next we encode the functions P in f_5, f_6. Finally, we encode the final XOR in f_7, f_8.

$$\begin{aligned} f_5 &= a_5 + a_4 \qquad & f_7 &= c_1 + a_5 + k_3 \\ f_6 &= a_6 + a_3 + a_4 + 1 \qquad & f_8 &= c_2 + a_6 + k_4 \end{aligned}$$

Altogether, the set of Boolean polynomials $\{f_1, \ldots, f_8\}$ in the indeterminates $p_1, p_2, c_1, c_2, k_1, \ldots, k_4, a_1, \ldots, a_6$ is an algebraic representation of the encryption process. △

2.8 Cryptanalysis and Fault Attacks

If an ANF or a CNF description of a cryptographic primitive is provided (such as in Example 2.38), it is fairly easy to mount an *algebraic attack* on the primitive. The details of such attacks are given in the next examples.

Example 2.39. Consider the problem KEY for an encryption function E. Find an encoding of the relation $E(k, p) = c$ in ANF (or in CNF). If the values p and c are known, substitute them into the encoding. Recovering the secret key k, and thus solving KEY, is nothing more than determining the set of zeros of the corresponding Boolean system (or the set of satisfying assignments of the corresponding CNF formula). △

Example 2.40. Consider the problem PREIMAGE for a hash function H. Find an encoding of the relation $H(m) = h$ in ANF (or in CNF). Substitute the value h in the encoding. Finding a corresponding message m', and thus solving PREIMAGE, is nothing more than determining the set of zeros of the corresponding Boolean system (or the set of satisfying assignments of the corresponding CNF formula). △

Strong cryptographic algorithms are designed in a way such that algebraic attacks, i.e. solving underlying Boolean systems or sets of clauses, are not feasible. However, if additional constraints are obtained, the attack may be successful. We describe a way how to construct such constraints by manipulating hardware implementations.

A *fault attack* is a special kind of *side-channel attack* where the attacker has access to the (hardware) implementation of the function and is able to intentionally inject physical disturbances during the operation. Fault attacks are very often realized by *differential fault analysis* (DFA) where the fault-free and the fault-affected calculations are compared. The fault corresponds to a difference in an intermediate state. The attack is conducted by symbolically propagating this difference towards the circuit outputs, which are known to the attacker.

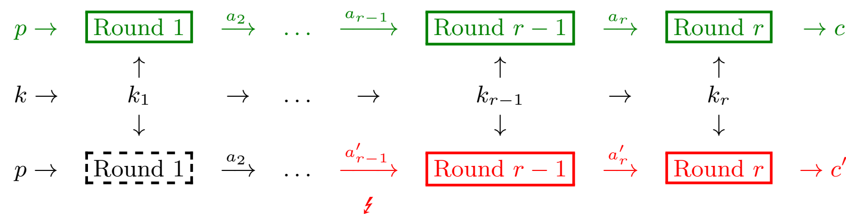
Fault attacks can be divided into various subcategories regarding the following criteria which depend on the capabilities of the attacker and the hardware implementation itself.

- *Targets.* E.g., an application-specific integrated circuit that realizes a cipher or a hash function, a microprocessor that runs cryptographic software, etc.
- *Means of physical fault-injections.* E.g., low-cost methods like glitching or over-heating vs. high-effort techniques such as electromagnetic or laser-based fault-injection which are more accurate.
- *Numbers of required faults.* The more faults are required to be injected by an attacker, the less applicable is the attack in practise.
- *Temporal accuracy.* This criterion reflects how accurate is the attacker w.r.t. the time progression, e.g. if it is possible to determine which part of the algorithm is currently carried out.
- *Spacial accuracy.* This criterion reflects how accurate is the attacker's knowledge about in which register or memory cells a certain intermediate state of the computation is stored.
- *Complexity of mathematical analysis.* The analysis is usually carried out by solving a set of constraints and checking a list of candidates. This step highly depends on the other criteria.
- *Total cost.* An estimation how much the attack costs (in terms of hardware, time, etc.).

A usual way how to apply faults attacks is to add so-called *fault equations* which describe the fault propagation to an instance created in a vanilla algebraic attack. (This approach is explained below in Example 2.41.) The addition of the fault equations should help a solver to find a solution. On one hand, one can manually taylor "ad-hoc" fault equations for a given cipher (e.g., see [112], [65]). On the other hand, there exist approaches, so-called *algebraic fault attacks* (AFA), where the fault equations are constructed in an automatic way. The later approaches benefit from the fact that more complicated fault relations can be generated than by a human-design approach.

In the next examples we provide a high-level description of an AFA on an iterative block cipher. Similar attacks can be launched on other cryptographic primitives that are based on iterative designs. In the first scenario, we extend an algebraic attack by fault equations derived from the last rounds.

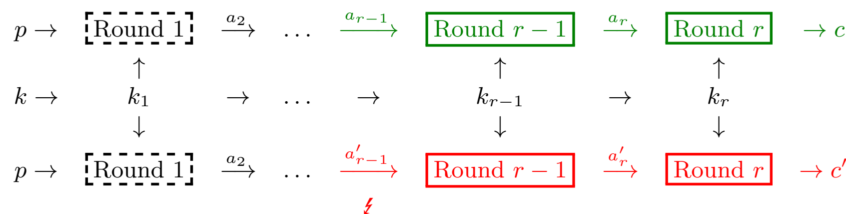
Example 2.41. The following diagram depicts an iterative block cipher under an algebraic fault attack. The first faulty-free encryption (i.e. $p \mapsto c$) corresponds to a vanilla algebraic attack. A fault is injected in the second encryption (i.e. $p \mapsto c'$) two rounds "before the end", i.e. in the state a'_{r-1}. The fault injection in the diagram is indicated by the lightning bolt. The key schedule is the same for both calculations.



The entire fault-free encryption (i.e. $p \mapsto c$) and the last two round of the faulty encryption (i.e. $a'_{r-1} \mapsto c'$) are encoded in ANF (or in CNF). Moreover, we may add special restrictions on the fault such as $a_{r-1} \oplus a'_{r-1} = \delta$ where the bit string δ is partially known, or $w_H(a_{r-1} \oplus a'_{r-1}) \leq d$ where w_H denotes the Hamming weight and $d \in \mathbb{N}$. We solve the whole instance for the unique keys k_{r-1} and k_r. By undoing the (bijective) key-schedule we obtain the unique key k. △

In the second scenario, we consider encoding only of the last rounds.

Example 2.42. In the setting of Example 2.41 consider the following diagram.



In contrast to Example 2.41, we encode only the last rounds for each encryption, i.e. we encode $a_{r-1} \mapsto c$ of the fault-free encryption and $a'_{r-1} \mapsto c'$ of the faulty encryption. Again, we may add some extra constraints on the fault as mentioned in Example 2.41. Solving the instance gives us the set of possible candidates for (k_{r-1}, k_r). For each candidate we undo the key-schedule and verify if the obtained key is correct. Of course, we can iterate the attack with various fault locations and obtain more restrictive constraints for the key candidates. △

Although real fault injections are done on hardware devices, in this thesis we are simulating the fault injection in software, i.e. we model fault injections by `XOR`ing random values to the intermediate state words.

Chapter 3

The Boolean Border Basis Algorithm

Motivation From the theoretic point of view, border bases are motivated from seeking vector space bases of P/I where I is a zero-dimensional ideal in a polynomial ring P. If such a basis forms an order ideal $\mathcal{O}$, and a set of polynomials $G \subset P$ has a special shape (namely, we have $\mathrm{Supp}(G) \subseteq \mathcal{O} \cup \partial\mathcal{O}$, and exactly one term in $\mathrm{Supp}(g)$ for $g \in G$ is contained in $\partial\mathcal{O}$), the set G is said to be an $\mathcal{O}$-border basis of I. Hence any polynomial with the support "outside" the border $\partial\mathcal{O}$ can be rewritten using G to a polynomial with the support completely in $\mathcal{O}$.

The *Border Basis Algorithm* (BBA) computes border bases that corresponds to special order ideals. (Later we shall see that such border bases generalize Gröbner bases.) The BBA is an iterative algorithm where an initial set of polynomials $V \subset P$ is updated in every iteration. The philosophy of the BBA can be outlined as follows.

- *Keep the degree under control.* Given a set of intermediate polynomials $V \subset P$, we form only linear syzygies $x_k f_i + x_\ell f_j$ in P with $f_i, f_j \in V$ during the algorithm, i.e., the degree is increased at most by one in one step. The newly derived polynomials are then appended to V and thus used for creating new linear syzygies.
- *Use linear algebra.* The goal of computing the linear syzygies is to cancel leading terms. The BBA uses linear LT_σ-interreductions of the intermediate polynomials and thus relies on linear algebra algorithms.
- *Keep the support under control.* We limit the growth of the support of the polynomials in V by a mechanism based on an order ideal U. Simply speaking, if the support of a polynomial in V is not contained in U, it is not used further in forming the linear syzygies. Thus $\mathrm{Supp}(V)$ contains at most $\#(U \cup x_1 U \cup \cdots \cup x_n U)$ different terms in each iteration. The universe U is enlarged carefully during the BBA.

Notice that the BBA overcomes some problems of the Gröbner basis algorithm (GBA, see Algorithm 2.1). Namely, the degree is not increased so greatly as during the GBA (recall that the GBA uses fundamental syzygies, i.e. S-polynomials), and only the linear reductions are used in the BBA, unlike the normal remainder reductions in the GBA.

In order to aply the theory of border bases to various benchmarks coming from computer science, border bases over Boolean polynomials become important. This motivates us to introduce *Boolean border bases* and the *Boolean Border Basis Algorithm* (BBBA) in this chapter. Moreover, we mention details on how to implement the BBBA in C++.

Related work From the historical point of view, first remarks how to compute such bases were given in [83]. A complete description follows in [70]. However, the order

ideals in [70] are restricted to a special form and depend on a term ordering. For a computation of border bases that do not depend on any term ordering, we refer the reader to [67]. The characterizations of border bases was given in [69]. The Boolean version of border bases was introduced in [58]. Since then, there have been attempts to generalize various improvements of GBA such as signature techniques (see [93]) to BBA. Some results in this direction are given in [59]. The only non-high-level implementation of the BBA known to the author is the one in `ApCoCoA` [109]. The first implementation of BBBA was given in [58]. This implementation in [58] that adopted the basic reasoning about Boolean polynomials to the BBA has been improved successively in [53, 55, 59]. This chapter is based on [53, 55, 58, 59].

Structure and contents The structure of this chapter is as follows. The definition of border bases and the BBA is recalled in Sect. 3.1 and 3.2. These sections contain a high-level translation of the BBA to the Boolean case. In the rest of the chapter we refine the BBBA described in Sect. 3.2 and provide various details and improvements such that the resulting description does not contain any gaps. To implement the Boolean BBA efficiently, we have to find suitable data structures for squarefree terms, order ideals and Boolean polynomials. Order ideals are discussed in Sect. 3.3, and linear interreductions are studied in Sect. 3.4 together with some implementation details in Sect. 3.5. Having described the main parts separately in the previous sections, we are able to restructure the BBBA in Sect. 3.6 such that the resulting description is very close to the actual C++ implementation. In Sect. 3.7 we outline possible improvements of the BBBA. In Sect. 3.8 we conclude the chapter with some timings which illustrate the feasibility of different versions of the BBBA, and we compare our implementation to other methods.

3.1 Border Bases

We start this section with recalling the general concept of border bases. In the following we let K be a field and $P = K[x_1, \ldots, x_n]$ a polynomial ring over K. One of the basic ideas of border bases is to find vector space bases of P/I that form order ideals. The following type of systems of generators of I allow us to rewrite all polynomials in terms of an order ideal $\mathcal{O}$.

Definition 3.1. Let $\mathcal{O} = \{t_1, \ldots, t_\mu\}$ be an order ideal, and let $\partial\mathcal{O} = \{b_1, \ldots, b_\nu\}$ be its border.

(a) A set of polynomials $G = \{g_1, \ldots, g_\nu\}$ is called an **$\mathcal{O}$-border prebasis** if $g_j = b_j - \sum_{i=1}^{\mu} c_{ij}\, t_i$ with $c_{1j}, \ldots, c_{\mu j} \in K$ for $j = 1, \ldots, \nu$.

(b) An $\mathcal{O}$-border prebasis $G \subset I$ is called an **$\mathcal{O}$-border basis** of I if the residue classes $\overline{\mathcal{O}} = \{t_1 + I, \ldots, t_\mu + I\}$ in P/I form a K-basis of P/I.

In the setting of (a) and (b), we sometimes say that the polynomial g_j is the border basis element of G which corresponds to the border term b_j. We recall some useful facts about border bases in the following proposition. The proofs can be found in [73, Sect. 6.4].

Proposition 3.2. *Let $\mathcal{O}$ be an order ideal in $\mathbb{T}^n$, let $I \subset P$ be a zero-dimensional ideal.*

(a) Assume that the residue classes of the elements of $\mathcal{O}$ form a K-vector space basis of P/I. Then there exists a unique $\mathcal{O}$-border basis of I.

(b) Let G be an $\mathcal{O}$-border basis of I. Then we have $I = \langle G \rangle$ and $P = I \oplus \langle \mathcal{O} \rangle_K$.

It is natural to ask whether the order ideals satisfying the condition in Claim (a) exist for any zero-dimensional ideal I. The answer is yes, and the order ideal is given by $\mathcal{O}_\sigma(I) = \mathbb{T}^n \setminus \mathrm{LT}_\sigma(I)$. This gives us one way of constructing an $\mathcal{O}$-border basis, namely, we can extend a Gröbner basis as in Example 3.3.

Example 3.3. Let I be a zero-dimensional ideal in P. Choose a term ordering σ, compute the reduced σ-Gröbner basis G of I, let $\mathcal{O}_\sigma(I) = \mathbb{T}^n \setminus \mathrm{LT}_\sigma(I) = \{t_1, \ldots, t_\mu\}$, and let $\partial\mathcal{O}_\sigma(I) = \{b_1, \ldots, b_\nu\}$. Then the set $G' = \{g_1, \ldots, g_\nu\}$, where $g_j = b_j - \mathrm{NF}_G(b_j)$ for $j = 1, \ldots, \nu$, is an $\mathcal{O}_\sigma(I)$-border basis of I which contains G (see [73, Prop. 6.4.18]). △

Conversely, every $\mathcal{O}_\sigma(I)$-border basis of I contains reduced σ-Gröbner basis of I. Thus $\mathcal{O}_\sigma(I)$-border bases are actually extensions of Gröbner bases because the order ideals are restricted to a special form (cf. [58, Ex. 2.3]). However, not every border basis is constructed by extending a reduced Gröbner basis, as the next example shows.

Example 3.4. Let $K = \mathbb{Q}$, let $P = K[x_1, x_2]$, and let $I = \langle x_1^2 + x_1x_2 + 1,\ x_2^2 + 2x_1x_2 + 1 \rangle$. Then $\mathcal{O} = \{1,\ x_1,\ x_2,\ x_1x_2\}$ is an order ideal and the residue classes of the terms in $\mathcal{O}$ form a K-basis of P/I. Notice that this order ideal is not of the form $\mathbb{T}^n \setminus \mathrm{LT}_\sigma(I)$ for any term ordering σ, since $x_1x_2 \in \mathrm{LT}_\sigma(I)$ for both cases $x_1 >_\sigma x_2$ and $x_2 >_\sigma x_1$. This order ideal has its border equal to $\partial\mathcal{O} = \{x_1^2,\ x_1^2x_2,\ x_2^2x_1,\ x_2^2\}$. In this setting the set $\{x_1^2 + x_1x_2 + 1,\ x_1^2x_2 + x_1 - x_2,\ x_1x_2^2 - x_1 + 2x_2,\ x_2^2 + 2x_1x_2 + 1\}$ is an $\mathcal{O}$-border basis of I which is not constructed from a Gröbner basis as in Example 3.3. △

Next we introduce the definition of a Boolean $\mathcal{O}$-border basis. As usual for the Boolean case, we let $K = \mathbb{F}_2$, $P = \mathbb{F}_2[x_1, \ldots, x_n]$, and $F = \langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$. A border basis of an ideal I in P which contains the field ideal has the following shape.

Proposition 3.5. *Let $\mathcal{O}$ be an order ideal in $\mathbb{T}^n$, and let I be an ideal in P which contains the field ideal F and has an $\mathcal{O}$-border basis.*

(a) We have $\mathcal{O} \subset \mathbb{S}^n$ and a disjoint union $\partial\mathcal{O} = (\partial\mathcal{O})^{\mathrm{sf}} \cup x_1\mathcal{O}_1 \cup \cdots \cup x_n\mathcal{O}_n$, where $\mathcal{O}_i$ is the set of all terms in $\mathcal{O}$ divisible by x_i.

(b) For $i \in \{1, \ldots, n\}$ and $t \in \mathcal{O}_i$, the border basis element corresponding to $x_i t$ is $x_i t + t$.

Proof. To prove (a), it suffices to prove the first claim. If a term in $\mathcal{O}$ is not squarefree, it is of the form $x_i^2 t$ with $t \in \mathcal{O}$. But then $\mathcal{O}$ is not linearly independent modulo I, since $x_i^2 t + x_i t$ is in I. To prove (b), we observe that the polynomial $x_i t + t$ is a multiple of $x_i^2 + x_i$ and hence in I. This implies the claim. □

In other words, Claim (b) tell us that the border basis elements corresponding to $\partial\mathcal{O} \setminus (\partial\mathcal{O})^{\,\mathrm{sf}}$ are trivial. This fact and Proposition 2.12 motivates us to the following definition.

Definition 3.6. Let $P = \mathbb{F}_2[x_1, \ldots, x_n]$, let $\mathcal{O} = \{t_1, \ldots, t_\mu\}$ be an order ideal in $\mathbb{S}^n$. Let $I \subseteq P$ be an ideal which contains the field ideal $F = \langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$.

(a) A set of polynomials $G \subset P$ is called a **Boolean $\mathcal{O}$-border prebasis** if G is $\mathcal{O}$-border prebasis.

(b) A Boolean $\mathcal{O}$-border prebasis $G \subset I$ is called a **Boolean $\mathcal{O}$-border basis** of I if G is an $\mathcal{O}$-border basis of I.

(c) Let G be a Boolean $\mathcal{O}$-border basis of I. A subset $G' \subset G$ in ANF is called a **short Boolean $\mathcal{O}$-border basis** of I if G' contains all border basis elements in G which correspond to the terms in $(\partial\mathcal{O})^{\,\mathrm{sf}}$.

Let us see an example of an order ideal $\mathcal{O}$ and an $\mathcal{O}$-border basis G which is not constructed from a Gröbner basis such that the ideal $I = \langle G \rangle$ contains the field ideal.

Example 3.7. In the polynomial ring $P = \mathbb{F}_2[x_1, x_2, x_3, x_4]$, consider the ideal $I = \langle x_2x_4{+}x_3x_4{+}1,\ x_1x_3{+}x_1x_2{+}1,\ x_1x_3x_4\rangle{+}F$ where $F = \langle x_1^2{+}x_1,\ x_2^2{+}x_2,\ x_3^2{+}x_3,\ x_4^2{+}x_4\rangle$ is the field ideal. Then one can check that I has a border basis for the order ideal $\mathcal{O} = \{1,\ x_1,\ x_2,\ x_3,\ x_4,\ x_1x_2,\ x_2x_3,\ x_3x_4,\ x_1x_4\}$. This border basis is not constructed from a Gröbner basis, since $x_2x_3 >_\sigma x_3x_4$ and $x_1x_3 >_\sigma x_1x_2$ cannot hold simultaneously. $\triangle$

3.2 The Border Basis Algorithms

In this section we present a high-level description of the BBA and the BBBA. As usual, P denotes a polynomial ring over the field K in the indeterminates $x_1, \ldots, x_n$. Let us start the section with the following useful notation.

Definition 3.8. For a set of polynomials $V \subset P$, we let $V^+ = V \cup x_1V \cup \cdots \cup x_nV$. This operation is called the **plus extension** of V.

Note that we use this notation even in the case when V is a set of terms or a vector subspace of P. In the BBA, the plus extension is used to enlarge a set of polynomials until a stable span is reached.

Definition 3.9. Let U be a set of terms in $\mathbb{T}^n$, and let V be a set of polynomials in P. Let $L = \langle U \rangle_K$ and $A = \langle V \rangle_K$. Define the following vector subspaces inductively as follows $A_0 = A$ and $A_{i+1} = A_i^+ \cap L$ for $i \in \mathbb{N}$. The union $A_\infty = \bigcup_{i=0}^{\infty} A_i$ is called the **L-stable span** of A.

Note that if the set V is LT_σ-interreduced, it holds $\big(\langle V \rangle_K\big)^+ = \langle V^+ \rangle_K$. Now we are ready to present the BBA in Algorithm 3.1.

The loop in Steps 4 – 12 of the algorithm computes the $\langle U \rangle_K$-stable span of $\langle V \rangle_K$ (see [70, Prop. 13]. The condition $\partial\mathcal{O} \subset U_{\mathrm{old}}$ is very important (see [70, Prop. 16]

Algorithm 3.1 `BBA` (The Border Basis Algorithm)

Input: Generators $\{f_1, \dots, f_s\}$ of a zero-dimensional ideal I and a term ordering σ.
Output: The $\mathcal{O}_\sigma(I)$-border basis of I.
Require: The algorithm `FinalReduction` for extracting the border basis from the last stable span described in [70, Prop. 17].

1: Let $U = \langle \mathrm{Supp}(f_1) \cup \cdots \cup \mathrm{Supp}(f_s) \rangle_{\mathrm{OI}}$.
2: Let V be an LT_σ-interreduced basis of $\langle f_1, \dots, f_s \rangle_K$.
3: **repeat**
4:   **repeat**
5:     Compute a set of polynomials W' such that $V \cup W'$ is an LT_σ-interreduced basis of $\langle V^+ \rangle_K$.
6:     **repeat**
7:       $W := \{w \in W' \mid \mathrm{LT}_\sigma(w) \in U\}$
8:       $U' := \langle \bigcup_{w \in W} \mathrm{Supp}(w) \setminus U \rangle_{\mathrm{OI}}$
9:       $U := U \cup U'$
10:     **until** $U' = \emptyset$
11:     $V := V \cup W$
12:   **until** $W = \emptyset$
13:   $\mathcal{O} := U \setminus \langle \mathrm{LT}_\sigma(V) \rangle$
14:   Let $U_{\mathrm{old}} := U$ and $U := U^+$.
15: **until** $\partial \mathcal{O} \subset U_{\mathrm{old}}$
16: Apply `FinalReduction`$(V, \mathcal{O})$ and return the result.

and ensures that such a border basis exists. The algorithm `FinalReduction` has a low time complexity and will be ignored in the following. Notice that we used already the improved version of the BBA described in [70, Prop. 21]. The order ideal U in this algorithm is called the (computational) **universe**. Since we are performing linear algebra operations in the vector space $\langle U^+ \rangle_K$, our goal is to keep it as small as possible at all times. For further details on this algorithm, we refer the reader to [70].

In the rest o the section, we translate the BBA to its Boolean version in a rather straightforward way, i.e. we let $K = \mathbb{F}_2$, $P = \mathbb{F}_2[x_1, \dots, x_n]$, and $F = \langle x_1^2 + x_1, \dots, x_n^2 + x_n \rangle$. While computing a border basis of an ideal containing the field ideal F, we can restrict everything to polynomials having only squarefree terms in their supports. To emphasize that the plus extension is done in the ring of Boolean polynomials $\mathbb{B}_n$, we use the brackets around the plus symbol, i.e., $V^{(+)} = V \cup x_1 V \cup \cdots \cup x_n V$ in $\mathbb{B}_n$. Similarly, for $t, t' \in \mathbb{S}^n$, we let $t \cdot t'$ be the product of t and t' followed by reduction modulo the field ideal F. For a polynomial $f \in P$, the normal form $\mathrm{NF}_F(f)$ is obtained by replacing each term in $\mathrm{Supp}(f)$ by its squarefree part. Putting these facts into the BBA in Algorithm 3.1, we get the following Boolean version in Algorithm 3.2.

Notice that, during the course of this algorithm, we always have $U \subset \mathbb{S}^n$ and $V \subset \langle \mathbb{S}^n \rangle_{\mathbb{F}_2}$. Since $\mathbb{T}^n$ contains $\binom{n+d-1}{d}$ terms of degree d, while $\mathbb{S}^n$ contains only $\binom{n}{d}$ terms of degree d, the universe and the resulting border basis will typically be much smaller

Algorithm 3.2 `BBBA` (The Boolean Border Basis Algorithm)

Input: Generators $\{f_1, \ldots, f_s\}$ of an ideal I that contain the field ideal F and a term ordering σ.
Output: A short Boolean $\mathcal{O}_\sigma(I)$-border basis of I.
Require: The algorithm `FinalReduction` for extracting the border basis from the last stable span described in [70, Prop. 17].

1: Let $U = \langle \mathrm{Supp}(\mathrm{NF}_F(f_1)) \cup \cdots \cup \mathrm{Supp}(\mathrm{NF}_F(f_s)) \rangle_{\mathrm{OI}}$.
2: Let V be an LT_σ-interreduced K-vector space basis of $\langle \mathrm{NF}_F(f_1), \ldots, \mathrm{NF}_F(f_s) \rangle_K$.
3: **repeat**
4: **repeat**
5: Compute a set of polynomials W' such that $V \cup W'$ is an LT-interreduced basis of $\langle V^{(+)} \rangle_K$.
6: **repeat**
7: $W := \{w \in W' \mid \mathrm{LT}_\sigma(w) \in U\}$
8: $U' := \langle \bigcup_{w \in W} \mathrm{Supp}(w) \setminus U \rangle_{\mathrm{OI}}$
9: $U := U \cup U'$
10: **until** $U' = \emptyset$
11: $V := V \cup W$
12: **until** $W = \emptyset$
13: $\mathcal{O} := U \setminus \mathrm{LT}_\sigma(V)$
14: Let $U_{\mathrm{old}} := U$ and $U := U^{(+)}$.
15: **until** $\partial \mathcal{O}^{\mathrm{sf}} \subset U_{\mathrm{old}}$
16: Apply `FinalReduction`$(V, \mathcal{O})$ and return the result.

for the Boolean BBA in comparison to the BBA.

3.3 Squarefree Terms and Their Order Ideals

The motivation for the next sections is to turn the high-level description given in Algorithm 3.2 into more exact instructions such that the new description is closer to the actual implementation. In this section, we look at the problem of implementing order ideals in $\mathbb{S}^n$ and the necessary order ideal operations efficiently. We start with representations of squarefree terms. There are two main operations that have to be considered.

(T1) Multiplication of a term by a term.

(T2) Deciding whether a term is divisible by a term.

Let $t = x_1^{\alpha_1} \cdots x_n^{\alpha_n} \in \mathbb{S}^n$ be a squarefree term, i.e. $\alpha_i \in \{0, 1\}$ for $i = 1, \ldots, n$. We have two options how to implement terms and operations (T1), (T2):

- The term t can be implemented in the *dense representation* via the bittuple $\alpha = (\alpha_1, \ldots, \alpha_n)$. The multiplication of terms $t \cdot t'$ in $\mathbb{S}^n$, i.e., multiplication

and subsequent reduction modulo the field ideal, can then be implemented via `OR` of the bittuples. Specifically, multiplication of t by an indeterminate x_i is nothing but an `OR` with 1 in the i-th position. Similarly, a term t with an exponent vector α divides a term t' with an exponent vector α' if and only if $(\alpha \texttt{ AND } \alpha') = \alpha$ holds.

- On the other hand, we can use a *sparse representation*, i.e. it suffices to store the sorted indices of the indeterminates appearing in the term. Multiplying t by an indeterminate x_i computed modulo $x_i^2 + x_i$ is implemented as inserting the index i into sorted array if i did not appear in the representation (in the correct position in order to preserve the sorted array). A term $t \in \mathbb{S}^n$ divides a term $t' \in \mathbb{S}^n$ if the indices in the representation of t are contained in the representation of t'.

The following example provide the difference between the two representations.

Example 3.10. Let $t = x_1 x_8$ be a term in $\mathbb{S}_{10}$. Then t is represented as $(1, 0, 0, 0, 0, 0, 0, 1, 0, 0)$ in the dense way and by $(1, 8)$ in the sparse way. $\triangle$

In practically relevant cases we may have a bound on the maximal degree of the terms that are used at some time during the computation. If the maximum degree is d, and we have $n = 2^k$ indeterminates, the sparse representation requires at most dk bits. In our setting, we usually have a large number of indeterminates, while the maximum degree is typically below 8. Thus we have $dk \ll 2^k = n$, and the sparse representation is thus more memory efficient and convenient for our purposes.

Let us give concrete implementation details of the squarefree terms. A squarefree term $t = x_{i_1} \cdots x_{i_k} \in \mathbb{S}^n$ with $1 \leq i_1 < \cdots < i_k \leq n$ can be implemented in the sparse way as the sorted `array` of `uint` $\tilde{t} = (i_1, \ldots, i_k)$. The arrays are allocated length $d \in \mathbb{N}$. The number d corresponds to the maximal degree occurring in the run of the algorithm, and it has to be determined before the run of the algorithm. Typically, d is set according to the amount of RAM memory available on a computer. To give a sense of scale, the maximal degree does not exceed 8 for the examples of quadratic systems in [46]. Thus we use only terms in $\mathbb{S}^n$ with degree at most d. This set is denoted by $\mathbb{S}^n_{\leq d}$.

The squarefree terms are ordered by a degree compatible term ordering σ because the BBBA is a degree-by-degree algorithm (see Definition 3.8). Moreover, we can rearrange the indeterminates according to how frequently they appear in the input, where x_1 is the most frequent. The rearrangements can speed up the BBBA in some cases.

The terms used in the algorithm are mapped to numbers such that these numbers reflect the term ordering σ.

Definition 3.11. Let σ be a term ordering. The unique bijective map $\psi_{\sigma,d} : \mathbb{S}^n_{\leq d} \to \{1, \ldots, \#\mathbb{S}^n_{\leq d}\} \subseteq \mathbb{N}$ with the property $\psi_\sigma(t) \leq \psi_\sigma(t')$ if and only if $t \leq_\sigma t'$ for $t, t' \in \mathbb{S}^n_{\leq d}$ is called the **numbering of terms** in $\mathbb{S}^n_{\leq d}$ induced by σ.

To illustrate the definition, we give the following example.

Example 3.12. Using $\sigma = \texttt{DegLex}$, $d = 2$ and $n = 2$, the map $\psi_{\sigma,d}$ defined by $1 \mapsto 1$, $x_2 \mapsto 2$, $x_1 \mapsto 3$, $x_1 x_2 \mapsto 4$ is a numbering of terms in $\mathbb{S}^n_{\leq d}$ induced by σ. $\triangle$

To implement order ideals in $\mathbb{S}^n$ (such as the universe U) efficiently, we represent them by their cogenerators (see Definition 2.3). Clearly, an arbitrary set of cogenerators can be transformed to a minimal one by removing multiples iteratively. Every set of cogenerators contains a unique minimal one. We shall represent order ideals by their unique minimal set of cogenerators. Thus, after every order ideal operation, we minimize the resulting set of cogenerators.

We consider the following order ideal operations and functions.

(O1) Membership of a term in an order ideal in $\mathbb{S}^n$.

(O2) Computing $U^{(+)}$ for an order ideal U in $\mathbb{S}^n$.

(O3) Calculating $U \setminus \mathrm{LT}_\sigma(V)$ for an order ideal $U \subset \mathbb{S}^n$ and a set of polynomials $V \subset \mathbb{B}_n$.

(O4) Determining whether $(\partial\mathcal{O})^{\mathrm{sf}} \subseteq U$ for order ideals $\mathcal{O}$ and U in $\mathbb{S}^n$.

(O1) can be checked by testing if the term divides one of the cogenerators. The order ideal membership problem is decided by Algorithm 3.3. Its proof of correctness follows immediately from the definition of cogenerators.

Algorithm 3.3 `IsInOI` (Order Ideal Membership Test)

Input: The minimal set of cogenerators C of an order ideal $\mathcal{O}$ in $\mathbb{S}^n$, $t \in \mathbb{S}^n$.
Output: `True` if $t \in \mathcal{O}$, `False` otherwise.

```
1: a := False
2: foreach c in C do
3:    if t divides c then
4:       a := True
5:    end if
6: end foreach
7: return a
```

The computation of (O2) is straightforward, since it suffices to take the union of the sets of cogenerators of U, x_1U, ..., x_nU, and to minimize. The following proposition shows how we can calculate cogenerators of an order ideal minus a monomial ideal. Altogether, we solve (O3).

Proposition 3.13. *Let U be an order ideal in $\mathbb{T}^n$, let C be a set of cogenerators of U, and let $t \in U$. Define the set $D = \{\frac{t'}{x_i} \mid i \in \{1,\dots,n\}$, $t' \in C$, t divides t', and x_i divides $t'\}$. Then $(C \cup D) \setminus \langle t \rangle$ is a set of cogenerators of the order ideal $U \setminus \langle t \rangle$.*

Proof. Let $u \in U \setminus \langle t \rangle$. Then u divides a cogenerator $t' \in C$. If t' is not a multiple of t, the claim is trivially true. If t' is a multiple of t, then u is a proper divisor of t' and hence a divisor of one of the terms $\frac{t'}{x_i}$ in D. □

Notice that we can replace the condition that "x_i divides t'" in the definition of D by "x_i divides t" if $U \subset \mathbb{S}^n$. We present Algorithm 3.5 (and its subroutine Algorithm 3.4) for computing the minimal set of cogenerators of an order ideal minus a monomial ideal.

The remaining task (O4), namely checking whether $(\partial\mathcal{O})^{\,\mathrm{sf}}$ is contained in U, is dealt with by the following proposition.

Proposition 3.14. *Let U be an order ideal in $\mathbb{S}^n$, let $\mathcal{O}$ be an order ideal contained in U, let C be the minimal set of cogenerators of $\mathcal{O}$, and let $D = x_1C \cup \cdots \cup x_nC$. Then we have $(\partial\mathcal{O})^{\,\mathrm{sf}} \subseteq U$ if and only if $D \subseteq U$.*

Proof. Clearly, the terms in D are contained in $\mathcal{O}$ or in $(\partial\mathcal{O})^{\,\mathrm{sf}}$, and therefore in U. Conversely, assume that $D \subseteq U$. Every term $t \in (\partial\mathcal{O})^{\,\mathrm{sf}}$ is of the form $t = x_it'$ with $i \in \{1,\ldots,n\}$ and $t' \in \mathcal{O}$. Thus there exists a cogenerator $u \in C$ such that $u = t't''$ for some $t'' \in \mathbb{S}^n$. Now the claim follows from $t''\,t = x_i \cdot u \in D \subseteq U$ and the fact that U is an order ideal. □

Notice that it is more complicated to compute the border of $\mathcal{O}$ itself. We have to multiply the cogenerators of $\mathcal{O}$ by indeterminates and consider all factors of the resulting squarefree terms in D. Since we do not need this method in Proposition 3.14, we merely indicate it by an example.

Example 3.15. In $\mathbb{S}^3$ we consider $\mathcal{O} = \{1,\, x_1,\, x_3,\, x_1x_3\}$, i.e. we have $\mathcal{O} = \langle x_1x_3\rangle_{\mathrm{OI}}$. By multiplying the cogenerator x_1x_3 with indeterminates, we get one squarefree border term, namely $x_1x_2x_3$. Now the divisors of this term provide the squarefree border $(\partial\mathcal{O})^{\,\mathrm{sf}} = \{x_2,\, x_3,\, x_1x_2,\, x_2x_3,\, x_1x_2x_3\}$. △

The test provided by Proposition 3.14 is easy to implement. Terms in the set D are produced by sequentially flipping zeros to ones in the exponents of the terms of C. Furthermore, in applications such as algebraic attacks, we usually end up with a very small order ideal. Therefore the final computation of the border is much less expensive than the computation of the borders of the intermediate order ideals $\mathcal{O}$ would be. Algorithm 3.6 decides if the squarefree border of one order ideal is contained in some other order ideal.

3.4 Linear Interreduction for Boolean Polynomials

Linear interreduction is a very important ingredient of the BBA. Hence, before moving to a restructured version of the BBBA, we discuss linear interreduction of Boolean polynomials. For a better understanding of linear LT_σ-interreduction, we formulate the following definitions which are analogous to the rewriting rules in the Gröbner basis theory (see [72, Def. 2.2.1]).

Definition 3.16. Let $V \subseteq \mathbb{B}_n$, and let $b, r, b' \in \mathbb{B}_n$.

(a) We say that b **linearly LT_σ-reduces to b' in one step** using r if $\mathrm{LT}_\sigma(b) = \mathrm{LT}_\sigma(r)$ and $b' = a + r$. We write $b \xrightarrow{r} b'$.

Algorithm 3.4 `OIminusTerm` (Order Ideal Minus a Monomial Ideal Generated by a Term)

Input: A term $t \in \mathbb{S}^n$, the minimal set of cogenerators C of an order ideal $\mathcal{O}$ in $\mathbb{S}^n$.
Output: The minimal set of cogenerators of the order ideal $\mathcal{O} \setminus \langle t \rangle$.

1: $D := \emptyset$
2: **foreach** c **in** C **do**
3:   **if** t divides c **then**
4:     **for** $i = 1$ **to** n **do**
5:       **if** x_i divides t **then**
6:         $D := D \cup \{\frac{c}{x_i}\}$
7:       **end if**
8:     **end for**
9:   **end if**
10: **end foreach**
11: Let A be the set of the minimal elements in $(C \cup D) \setminus \langle t \rangle$ w.r.t. division.
12: **return** A

Algorithm 3.5 `OIminusMI` (Order Ideal Minus a Monomial Ideal)

Input: The minimal set of cogenerators C' of an order ideal U in $\mathbb{S}^n$, a set of squarefree terms T.
Output: The minimal set of cogenerators C of the order ideal $U \setminus \langle T \rangle$.
Require: Algorithm 3.4.

1: $C := C'$
2: **foreach** t **in** T **do**
3:   **if** $t \in \langle C' \rangle_{\mathrm{OI}}$ **then**
4:     $C :=$`OIminusTerm`(t, C)
5:   **end if**
6: **end foreach**
7: **return** C

(b) We say that b **linearly** LT_σ**-reduces to** b' using V if there exist $v_i \in V$ for $i = 1, \ldots, k$ and $b_1, \ldots, b_{k-1} \in \mathbb{B}_n$ such that $b \xrightarrow{v_1} b_1 \xrightarrow{v_2} \ldots \xrightarrow{v_{k-1}} b_{k-1} \xrightarrow{v_k} b'$. We write $b \xrightarrow{V} b'$.

(c) A polynomial b with the property that there is no $r \in V$ such that $b \xrightarrow{r} b'$ for some $b' \in \mathbb{B}_n$ is called **linearly** LT_σ**-irreducible** with respect to V.

Obviously, we have $b \xrightarrow{b} 0$ for any polynomial b. The following example shows us that the result of a sequence of linear LT_σ-reductions is not uniquely determined in general.

Example 3.17. Let $V = \{x_1x_2 + 1,\, x_1x_2,\, x_1 + 1\} \subseteq \mathbb{B}_2$. Then $x_1x_2 + x_1 \xrightarrow{x_1x_2+1} x_1 + 1 \xrightarrow{x_1+1} 0$, and thus $x_1x_2 + x_1 \xrightarrow{V} 0$. On the other hand, $x_1x_2 + x_1 \xrightarrow{x_1x_2} x_1 \xrightarrow{x_1+1} 1$, and thus $x_1x_2 + x_1 \xrightarrow{V} 1$. △

Algorithm 3.6 `CheckBorder` (Checking the Border)

Input: Cogenerators C' of an order ideal U in $\mathbb{S}^n$, cogenerators D of an order ideal $\mathcal{O}$ in $\mathbb{S}^n$.

Output: `True` if $(\partial\mathcal{O})^{\mathrm{sf}} \subseteq U$, `False` otherwise; a set of squarefree terms C such that $\langle C\rangle_{\mathrm{OI}} = \langle C' \cup (\partial\mathcal{O})^{\mathrm{sf}}\rangle_{\mathrm{OI}}$.

1: $a :=$ `True`
2: $B := \emptyset$
3: **foreach** d **in** D **do**
4:    **for** $i = 1$ **to** n **do**
5:       $d' := x_i d$
6:       **if** $d' \in \mathbb{S}^n$ **and** $d' \notin \langle C'\rangle_{\mathrm{OI}}$ **then**
7:          $B := B \cup \{d'\}$
8:          $a :=$ `False`
9:       **end if**
10:    **end for**
11: **end foreach**
12: Let C be the set of the minimal elements in $(C' \cup B)$ w.r.t. division.
13: **return** (a, C)

If we would like to have unique linear LT_σ-reducers (and hence unique linear LT_σ-reductions), the set V has to be linearly LT_σ-interreduced.

Proposition 3.18. *Let V be a linearly LT_σ-interreduced set of Boolean polynomials. Let $b, b' \in \mathbb{B}_n$ such that $b \xrightarrow{V} b'$. Then the polynomial b' is uniquely determined.*

Proof. There exists exactly one element $v_1 \in V$ such that $\mathrm{LT}_\sigma(b) = \mathrm{LT}_\sigma(v_1)$, because the leading terms of the elements of V are pairwise distinct. Let $b_1 = b - v_1$. We have $b \xrightarrow{v_1} b_1$. There exists at most one element $v_2 \in V$ such that $\mathrm{LT}_\sigma(b_1) = \mathrm{LT}_\sigma(v_2)$. If there is no such v_2, the element b_1 is the unique linear LT_σ-reduction of b. Otherwise, we continue with $b_2 = b_1 - v_2$ in the same way, and the result follows by induction. □

The following proposition gives us another useful property of a linearly LT_σ-interreduced set of Boolean polynomials.

Proposition 3.19. *Let V be a linearly LT_σ-interreduced set of Boolean polynomials, and let $b, r \in \mathbb{B}_n$. Then we have $b \xrightarrow{V} 0$ if and only if $b \in \langle V\rangle_{\mathbb{F}_2}$.*

Proof. First we prove "⇒". By definition, there exist $v_1, \ldots, v_k \in V$ and $b_i \in \mathbb{B}_n$ for $i = 1, \ldots, k-1$ such that $b \xrightarrow{v_1} b_1 \xrightarrow{v_2} \ldots \xrightarrow{v_{k-1}} b_{k-1} \xrightarrow{v_k} 0$. Hence we get $b = v_1 + \cdots + v_k$ in $\mathbb{B}_n$, and henceforth $b \in \langle V\rangle_{\mathbb{F}_2}$.

Conversely, let $b = v_1 + \cdots + v_k$ for some pairwise distinct elements $v_1, \ldots, v_k$ in V. Because V is linearly LT_σ-interreduced, there exists a unique index $i_1 \in \mathbb{N}$ with $1 \le i_1 \le k$ such that $\mathrm{LT}_\sigma(v_{i_1}) = \mathrm{LT}_\sigma(b)$. Hence $b \xrightarrow{v_{i_1}} (b - v_{i_1})$. There exists a unique $i_2 \in \mathbb{N}$ with $1 \le i_2 \le k$ such that $\mathrm{LT}_\sigma(v_{i_2}) = \mathrm{LT}_\sigma(b - v_{i_1})$. By induction, we

create a zero linear LT_σ-reduction chain starting from b and having linear LT_σ-reducers $v_{i_1}, \dots, v_{i_k} \in V$. □

The above proposition does not hold for sets of Boolean polynomials that are not LT_σ-interreduced. We illustrate this fact in the next example.

Example 3.20. Let $V = \{x_1x_2 + x_1,\ x_1x_2 + x_2\} \subseteq \mathbb{B}_2$. We can see that $x_1 + x_2 \in \langle V \rangle_{\mathbb{F}_2}$, but $x_1 + x_2$ is linearly LT_σ-irreducible with respect to V. △

We are now ready to introduce and analyze Algorithm 3.7 for computing successive extensions of linearly LT_σ-interreduced sets. Algorithm 3.7 will be applied in Algorithm 3.10 in the next section.

Algorithm 3.7 `Reduce` (Extension of Linearly LT_σ-Interreduced Set)

Input: A non-zero Boolean polynomial b', a linearly LT_σ-interreduced set of Boolean polynomials V', and a degree compatible term ordering σ.

Output: A set $V \subseteq \mathbb{B}_n$ such that V is linearly LT_σ-interreduced and $\langle V \rangle_{\mathbb{F}_2} = \langle V' \cup \{b'\} \rangle_{\mathbb{F}_2}$.

1: $b := b', V := V'$
2: **while** there exists $r \in V$ with $\mathrm{LT}_\sigma(r) = \mathrm{LT}_\sigma(b)$ **do**
3: $\quad b := b + r$
4: **end while**
5: **if** $b \neq 0$ **then**
6: $\quad V := V \cup \{b\}$
7: **end if**
8: **return** V

Proposition 3.21. *Algorithm 3.7 returns a linearly* LT_σ*-interreduced list* V *such that* $\langle V \rangle_{\mathbb{F}_2} = \langle V' \cup \{b'\} \rangle_{\mathbb{F}_2}$ *holds.*

Proof. In Step 2 we search for a unique polynomial in V' which has the same leading term as b. If such a polynomial does not exist, the polynomial b is appended to V in Step 6.

The linear LT_σ-reduction chain is constructed in Steps 2–4. If $b' \xrightarrow{V'} 0$, then $b' \in \langle V' \rangle_{\mathbb{F}_2} \subseteq \langle V \rangle_{\mathbb{F}_2}$ by Proposition 3.19. If we have $b' \xrightarrow{V'} b \neq 0$, then we have $b' \in V$ by Step 6. □

3.5 Implementation of Boolean Polynomials and Linear Interreduction

After dealing with the combinatorial part of the Boolean BBA, we now turn to the implementation of polynomials and polynomial linear algebra. Clearly, this is the true core of the algorithm and needs to be optimized the most.

A set of polynomials V can be represented by a coefficient matrix whose columns are labeled with the terms in the support of V (see Definition 2.7). Moreover, the columns are sorted w.r.t. a term ordering σ. This is particularly useful, because the Boolean BBA is at its core a linear algebra algorithm for which the computation of the basis extension is the most demanding task. In our case, the matrices are Boolean, i.e. they consist of $0, 1$. We consider the following tasks, when implementing systems of polynomials.

(P1) Multiplying a Boolean polynomial by an indeterminate.

(P2) Addition of two Boolean polynomials.

(P3) Appending a new Boolean polynomial f to a set of Boolean polynomials V. (Note that $\mathrm{Supp}(f)$ may contain new terms that are not involved in $\mathrm{Supp}(V)$.)

(P4) Linear LT_σ-interreducing of a set of Boolean polynomials.

Again, there are two options how to implement coefficient matrices.

- The *dense representation* of a matrix contains all coefficients of the matrix.
- The *sparse representation* of a matrix contains only positions of non-zero entries of the matrix.

The following example provides the difference of the two representations.

Example 3.22. Let $t_1, \ldots, t_5$ be terms in $\mathbb{S}^n$ such that $t_1 >_\sigma t_2 >_\sigma \cdots >_\sigma t_5$ w.r.t. some term ordering σ. The following dense matrix represents the polynomials $f_1, \ldots, f_5 \in \mathbb{B}_n$.

$$\begin{array}{c} \begin{matrix} t_1 & t_2 & t_3 & t_4 & t_5 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \begin{array}{l} \\ \leftrightarrow f_1 = t_1 + t_2 + t_3 + t_4 + t_5 \\ \leftrightarrow f_2 = t_2 + t_5 \\ \leftrightarrow f_3 = t_3 + t_5 \\ \leftrightarrow f_4 = t_4 + t_5 \\ \leftrightarrow f_5 = t_5 \end{array}$$

The sparse representation w.r.t. the tuple of terms $(t_1, \ldots, t_5)$ is given below

$$\begin{array}{lcl} (1,2,3,4,5) & \leftrightarrow & f_1 = t_1 + t_2 + t_3 + t_4 + t_5 \\ (2,5) & \leftrightarrow & f_2 = t_2 + t_5 \\ (3,5) & \leftrightarrow & f_3 = t_3 + t_5 \\ (4,5) & \leftrightarrow & f_4 = t_4 + t_5 \\ (5) & \leftrightarrow & f_5 = t_5 \end{array}$$

△

Let us describe an approach that we chosed in our C++ implementation. A Boolean polynomial $g = t_1 + \cdots + t_k$, where $t_1 <_\sigma t_2 <_\sigma \cdots <_\sigma t_k$ are terms in $\mathbb{S}^n_{\leq d}$, is represented

as a `vector<int128_t>` via $\tilde{g} = \big(\psi_{\sigma,d}(t_1), \ldots, \psi_{\sigma,d}(t_k)\big) \in \mathbb{N}^k$, where $\psi_{\sigma,d}$ is the numbering of terms induced by σ (see Definition 3.11). Note that we can choose a different map instead of $\psi_{\sigma,d}$, but then accessing the leading term would have linear complexity instead of the constant one.

Example 3.23. Using $\psi_{\sigma,d}$ from Example 3.12, we represent $g = 1 + x_2 + x_1x_2 \in \mathbb{B}_2$ as the vector $\tilde{g} = (1, 2, 4)$. △

First of all, we look at (P1). Multiplication of g by an indeterminate x_i in $\mathbb{B}_n$ is done by translating $\tilde{g}$ back to terms in the support of g via $\psi^{-1}_{\sigma,d}$ and handled there. The result is then converted back using $\psi_{\sigma,d}$. These conversions do not slow down the overall performance according to our profiling. Moreover, the conversions are cached. The BBBA spends more than 95% in addition of two Boolean polynomials (P2), and this is very fast in this representation, namely only a symmetric difference of two sorted vectors.

Example 3.24. Using $\psi_{\sigma,d}$ from Example 3.12, we represent $f = 1 + x_2 + x_1x_2 \in \mathbb{B}_2$ as the vector $\tilde{f} = (1, 2, 4)$ and $g = 1 + x_1 + x_1x_2 \in \mathbb{B}_2$ as $\tilde{g} = (1, 3, 4)$. The sum $f + g = x_2 + x_1$ in $\mathbb{B}_2$ corresponds to $(2, 3)$. △

A system of Boolean polynomials V is implemented as a `vector<SparseRow>` $\tilde{V}$, where the inner collections represent individual Boolean polynomials. At this point, we may remark that the numbering of terms from Definition 3.11 can be alternatively defined only for the terms appearing in the system, and hence the representation would consist of smaller numbers. However, the representation of the entire system would have to be rewritten every time when a new term is introduced. In our case, the operation (P3) is very straight-forward. Remark that our relative sparse representation is efficient. E.g., in the dense representation, the coefficient matrix has to be dynamic, because we add new rows and new columns during the computation of $V^{(+)}$. Moreover, when we add a new column, i.e., when we introduce a new term, the resulting columns must be ordered via the term ordering from the biggest to the smallest term.

The operation (P4) is based on *Gaußian elimination* (GE) of the coefficient matrix, more precisely, on computing a *row echelon form* (REF) of the matrix. In the BBBA we consider a special problem of computing a REF, because we need to know the permutation of the rows (if some swapping has occurred), and many pivot positions are known beforehand. In fact, we are really computing a REF extension. The reducers are cached on-the-fly such that the polynomial that has the same leading term as a given polynomial is easily accessible. This idea is captured in the next definition.

Definition 3.25. Let $V \subseteq \mathbb{B}_n$ be an LT_σ-interreduced set of Boolean polynomials. A map $\varrho : \mathbb{S}_n \to V \cup \{\square\}$ such that

$$\varrho(t) = \begin{cases} v \in V & \text{if there exists } v \in V \text{ with } \mathrm{LT}_\sigma(v) = t, \\ \square & \text{otherwise.} \end{cases}$$

holds for $t \in \mathbb{S}_n$ is called a **map of reducers** of V.

It has turned out that GE without swapping rows, and optimized to minimize the writing all over the matrix, is the most suitable among different variants of GE. Let us compute the extension $\langle V\rangle_{\mathbb{F}_2} \subseteq \langle V^{(+)}\rangle_{\mathbb{F}_2}$ for a set $V \subset \mathbb{B}_n$. We start with the coefficient matrix $\mathrm{CM}_\sigma(V)$ which is already in REF. Then we append new rows from $\mathrm{CM}_\sigma(V^{(+)})$. We reduce each of these rows by rows in V. At the end we get either a new pivot, and we enlarge V, or we get a zero row which we cancel, and we proceed to the next row. Therefore we read only from V and write only in a single row at each step.

We prefer choosing pivots coming from rows in V. Otherwise, we could eliminate a row in V by a row in $V^{(+)} \setminus V$, and thus obtain a bigger extension than necessary. If the matrix is in the dense representation, addition of polynomials is nothing but a `XOR` of two rows. However, when we use a sparse representation, i.e., when only the non-zero position in a row are remembered, addition of polynomials results in the symmetric difference of two lists.

To illustrate how a system is LT_σ-interreduced and how Definition 3.25 is used, we present the following example.

Example 3.26. With the setting of Example 3.12, we want to LT_σ-interreduce $W = \{f, g, h\}$ with $f = 1+x_2+x_1x_2$, $g = 1+x_1+x_1x_2$ and $h = 1+x_1$ in $\mathbb{B}_2$. We initialize $\tilde{V} = \emptyset$ and $\varrho = \big(1 : \square \mid x_1 : \square \mid x_2 : \square \mid x_1x_2 : \square\big)$. Starting with f, we define $\tilde{V} = \big((1,2,4)\big)$ and $\varrho = \big(1 : \square \mid x_1 : \square \mid x_2 : \square \mid x_1x_2 : f\big)$. We continue with g as in Example 3.24 and get $\tilde{V} = \big((1,2,4),(2,3)\big)$ and $\varrho = \big(1 : \square \mid x_1 : x_2 + x_1 \mid x_2 : \square \mid x_1x_2 : f\big)$. Finally, we reduce h, and we get $\tilde{V} = \big((1,2,4),(2,3),(1,2)\big)$ and $\varrho = \big(1 : \square \mid x_1 : x_2 + x_1 \mid x_2 : 1 + x_2 \mid x_1x_2 : f\big)$. The tuples in $\tilde{V}$ correspond to the following polynomials $1 + x_2 + x_1x_2$, $x_2 + x_1$ and $1 + x_2$. $\triangle$

Note that the support and the cogenerators of the universe describes different aspects of the BBBA. E.g., during the computation of $V^{(+)}$, the support changes, since it is enlarged via multiplications by indeterminates, but the cogenerators remain untouched. Conversely, when the universe is enlarged, i.e., the cogenerators are changed, but the support remains the same.

3.6 The BBBA Refined

Let us recall the idea of the BBBA using the problem of finding the $\mathbb{F}_2$-rational solutions of a Boolean system $f_1 = \cdots = f_s = 0$. Let $V = \{f_1, \ldots, f_s\} \subseteq \mathbb{B}_n$. Define the ideal I that is generated by V and the field ideal. Suppose that the system has a unique $\mathbb{F}_2$-rational solution. (For instance, this is common in the scenario of algebraic attacks.) We are looking for a set of linear polynomials $G \subseteq I$ such that G is a linearly LT_σ-interreduced basis of $\langle G\rangle_{\mathbb{F}_2}$ and $\#\,\mathrm{Supp}(G) = \#G$. Hence the goal is to create new linearly independent linear polynomials in I and to keep the support of polynomials in the system as small as possible at the same time.

Given a set of Boolean polynomials $V = \{f_1, \ldots, f_s\}$, the BBBA generates new polynomials by forming and linearly LT_σ-interreducing $V^{(+)} = V \cup x_1V \cup \cdots \cup x_nV$. Every iteration of $V^{(+)}$ is then followed by linear LT_σ-interreduction. One could repeat these two operations in order to obtain the desired basis. However, this approach clearly leads

to an exponentially large amount of work since all polynomials in V are multiplied by n indeterminates.

Thus the operation $V^{(+)}$ in the BBBA is restricted by the order ideal U. The universe U is initially cogeneratored by $\bigcup_i \operatorname{Supp}(f_i)$. The $V^{(+)}$ operation is restricted to the polynomials that have their support contained in the universe. In this way, the growth of V and the support of the polynomials in V is lower. The universe is extended by the support of polynomials that have leading terms contained in U. This extension of the universe is described in Algorithm 3.8. (This algorithm will be used later in Algorithm 3.10.) The proof of correctness of Algorithm 3.8 is easy, and it is omitted.

Algorithm 3.8 `ExtendU` (An Extension of the Universe)

Input: Cogenerators C' of an order ideal U in $\mathbb{S}^n$, a linearly LT_σ-interreduced set of Boolean polynomials V.

Output: A set of cogenerators $C \supseteq C'$ such that $\mathrm{LT}_\sigma(f) \in \langle C \rangle_{\mathrm{OI}}$ for $f \in V$ implies that $\operatorname{Supp}(f) \subseteq \langle C \rangle_{\mathrm{OI}}$.

1: $C := C'$
2: **repeat**
3: $\quad D := C$
4: $\quad$ **foreach** f **in** V **do**
5: $\quad\quad$ **if** $\mathrm{LT}_\sigma(f) \in \langle C \rangle_{\mathrm{OI}}$ **and** f is not contained in $\langle C \rangle_{\mathrm{OI}}$ **then**
6: $\quad\quad\quad$ Let A be the set of the minimal cogenerators of $\langle C \cup \operatorname{Supp}(f) \rangle_{\mathrm{OI}}$.
7: $\quad\quad\quad$ $C := A$
8: $\quad\quad$ **end if**
9: $\quad$ **end foreach**
10: **until** $\#D = \#C$
11: **return** C

The successive computation of $V^{(+)}$ tends to repeat the consideration of multiples of polynomials that have been already multiplied by all indeterminates. To avoid this overhead, we introduce the following notion.

Definition 3.27. A Boolean polynomial $f \in \mathbb{B}_n$ is said to be **covered** in a linearly LT_σ-interreduced set $V \subseteq \mathbb{B}_n$ if $x_i f \xrightarrow{V} 0$ for all $i \in \{1, \ldots, n\}$.

Covered polynomials should be avoided because they do not introduce any new leading terms. The definition is equivalent to the condition $x_i f \in \langle V \rangle_{\mathbb{F}_2}$ for $i = 1, \ldots, n$ by Proposition 3.19. Checking the latter condition is quite expensive for large sets V. That is why we remember the polynomials that have been worked on as in the following example.

Example 3.28. Let $f = x_1x_2 + 1 \in \mathbb{B}_2$ and $V' = \{f\} \subseteq \mathbb{B}_2$. Let us compute $V'^{(+)}$ iteratively with successive linear LT_σ-interreduction. We compute $x_1 f = x_1x_2 + x_1 \xrightarrow{f} x_1 + 1$ and $x_2 f = x_1x_2 + x_2 \xrightarrow{f} x_2 + 1$. We get $V = \{x_1x_2 + 1,\, x_1 + 1,\, x_2 + 1\}$. Then f is covered in V, and therefore multiplication of $x_1x_2 + 1$ by indeterminates (once again)

does not yield new linearly independent polynomials during the computation of $V^{(+)}$. Thus we remember that the polynomial f is covered in V. $\triangle$

Algorithm 3.9 computes $\{b\}^{(+)}$ for a Boolean polynomial b and immediately linearly LT_σ-reduces the result against the known polynomials in V. To keep the pseudo-code simple, the covered polynomials that are easily discoverable are stored in the set $M \subseteq V$. The proof of correctness of Algorithm 3.9 follows directly from Proposition 3.21.

Algorithm 3.9 `PlusAndReduce` (Plus Extension and LT_σ-Interreduction)

Input: A non-zero Boolean polynomial b, a linearly LT_σ-interreduced set of Boolean polynomials V', a degree compatible term ordering σ, cogenerators C of an order ideal U in $\mathbb{S}^n$, and a set of polynomials $M' \subseteq V$ which are covered in V.
Output: A linearly LT_σ-interreduced set V such that $\langle V' \cup \{x_1 b, \dots, x_n b\}\rangle_{\mathbb{F}_2} = \langle V \rangle_{\mathbb{F}_2}$ if $\mathrm{Supp}(b) \subseteq \langle C \rangle_{\mathrm{OI}}$, $V = V'$ otherwise, and a set of covered polynomials M.
Require: Algorithm 3.7

1: $V := V'$, $M := M'$
2: **if** b is contained in $\langle C \rangle_{\mathrm{OI}}$ and $b \notin M$ **then**
3: **for** $i = 1$ **to** n **do**
4: $b' := x_i b$
5: Update V by calling `Reduce`(b', V, σ).
6: **end for**
7: $M := M \cup \{b\}$
8: **end if**
9: **return** (V, M)

Now we are able to describe a restructured version of the BBBA in Algorithm 3.10. Its subroutine `FinalReduction` refers to the algorithm in [70, Prop. 17] whose purpose is to extract the desired border basis from $\langle V \rangle_{\mathbb{F}_2}$. Notice that this algorithm can be easily modified to output only the polynomials having squarefree border terms.

As a pivoting strategy for the reduction process in the algorithm, we consider Boolean polynomials of the smallest degree and among them the ones having the smallest support, i.e. we will use the ordering which is given in the next definition.

Definition 3.29. Let $f, g \in \mathbb{B}_n$. We write $f \prec g$ if and only if $\deg(f) < \deg(g)$, or $\deg(f) = \deg(g)$ and $\#\,\mathrm{Supp}(f) < \#\,\mathrm{Supp}(g)$.

Proposition 3.30. *In the setting of Algorithm 3.10, Algorithm 3.10 outputs the Boolean $\mathcal{O}_\sigma(I)$-border basis of I.*

Proof. It is sufficient to prove that Algorithm 3.10 is equivalent to Algorithm 4.3 in [58]. Let the set V_a denote the set of all polynomials in V such that $\mathrm{Supp}(V_a) \subseteq U = \langle C \rangle_{\mathrm{OI}}$ where U is the current universe. Note that V may contain polynomials whose supports are not contained in $\langle C \rangle_{\mathrm{OI}}$. Thus the set V in Algorithm 4.3 in [58], corresponds to V_a.

The only difference in the initialization (apart from defining the new set M) occurs in Steps 3–5. They are equivalent to linear LT_σ-interreducing of the initial generators V.

Algorithm 3.10 `BBBA2` The BBBA (Restructured Version)

Input: A set of polynomials $V = \{f_1, \dots, f_s\} \subseteq \mathbb{B}_n$ such that $V \cup F$ generates a 0-dimensional ideal I and a degree compatible term ordering σ.
Output: A short Boolean $\mathcal{O}_\sigma(I)$-border basis of I.
Require: Algorithms 3.5, 3.6, 3.7, 3.8, 3.9, `FinalReduction`.

1: $V := \emptyset, M := \emptyset$
2: Let C be a set of the minimal cogenerators of the order ideal $\langle \bigcup_{i=1}^{s} \mathrm{Supp}(f_i)\rangle_{\mathrm{OI}}$.
3: **for** $i = 1$ **to** s **do**
4:    Update V by calling `Reduce`(f_i, V, σ).
5: **end for**
6: **repeat**
7:    **repeat**
8:       $V' := V$
9:       **foreach** f chosen in the increasing order according to "$\prec$" **in** V' **do**
10:          Update (V, M) by calling `PlusAndReduce`(f, V, σ, C, M).
11:       **end foreach**
12:       $C :=$`ExtendU`(C, V).
13:    **until** $\#V = \#V'$
14:    $D :=$`OIminusMI`$(C, \mathrm{LT}_\sigma(V))$.
15:    Update (a, C) by calling `CheckBorder`(C, D).
16: **until** $a =$ `True`
17: Apply `FinalReduction`$(V, \langle D\rangle_{\mathrm{OI}})$ and return the result.

Now we would like to show that Steps 7–13 computes the $\langle C\rangle_{\mathrm{OI}}$-stable span of V_a, i.e. that $\langle V_a\rangle_{\mathbb{F}_2} = \langle V_a^{(+)}\rangle_{\mathbb{F}_2} \cap \langle U\rangle_{\mathbb{F}_2}$ holds in Step 14. The inclusion "$\subseteq$" is trivial. Let us look at the other inclusion. The set M contains polynomials in V such that $M^{(+)} \subseteq \langle V\rangle_{\mathbb{F}_2}$, so elements in M can be omitted in Algorithm 3.9.

Let $U = \langle C\rangle_{\mathrm{OI}}$ and $v \in \langle V_a^{(+)}\rangle_{\mathbb{F}_2} \cap \langle U\rangle_{\mathbb{F}_2}$ in Step 14. We know that $v \xrightarrow{V} 0$ because $\langle V_a^{(+)}\rangle_{\mathbb{F}_2} \subseteq \langle V\rangle_{\mathbb{F}_2}$ after Step 11. This means that $v \in \langle V\rangle_{\mathbb{F}_2}$ by Proposition 3.19 because V is linearly LT_σ-interreduced to zero. We would like to show that $v \xrightarrow{V_a} 0$, which is equivalent to $v \in \langle V_a\rangle_{\mathbb{F}_2}$. Let $v = v_1 + \cdots + v_k$, where $\{v_1, \dots, v_k\} \subseteq V$ is a linearly LT_σ-interreduced set. Then $\mathrm{LT}_\sigma(v) = \mathrm{LT}_\sigma(v_i)$ for some $1 \le i \le k$. Since $\mathrm{LT}_\sigma(v_i) = \mathrm{LT}_\sigma(v) \in U$, we get $v_i \in \langle U\rangle_{\mathbb{F}_2}$, i.e. $v_i \in V_a$ after Step 12. We continue with the polynomial $v - v_i$, and we get that $\{v_1, \dots, v_k\} \subseteq V_a$ by induction.

The loop in Steps 9–11 enlarges V by elements in $\langle V_a^{(+)}\rangle_{\mathbb{F}_2}$ such that updated V is linearly LT_σ-interreduced. (This is equivalent to Step 5 of Algorithm 4.3 in [58].) Step 12 enlarges the universe in the same way as Steps 6–10 of Algorithm 4.3 in [58] do.

The rest (i.e., Steps 15–17) continues in the same way as Steps 13-16 of Algorithm 4.3 in [58] do. □

3.7 Improvements of the BBBA

In this section we outline possible further directions for improving Algorithm 3.10. The improvements are listed below.

- *Substitutions.* The original algorithm can be extended by substituting linear polynomials of the form x_i, $x_i + x_j$, $x_i + x_j + 1$, $x_i + x_j$ into the derived polynomials, whenever these special polynomials are found in V. E.g., if $x_i + x_j + 1$ with its leading term x_i is found in the ideal, we rewrite all polynomials containing x_i found so far using the rewriting rule $x_i \mapsto x_j + 1$. Thus we reduce the number of indeterminates in the system by one.

- *Extensions of the universe.* Enlarging U by $U^{(+)}$ or by its border may have the effect that the support of V blows up. The main idea of this improvement is to extend the universe as little as possible (e.g. see [70, Cor. 23]). Unfortunately, proving correctness of the BBBA when such approaches are applied remains an open problem.

- *Heuristics.* Tackling really hard problems, we may allow ourself to use some heuristics even though the algorithm may become incomplete. For instance, we can relax the ordering of terms in the coefficient matrix w.r.t. σ (e.g. see [67]). We may use random decisions in order to select the polynomial for the next reduction. Alternatively, we can interreduce only the sparse part of the system, and the dense part remains inactive, etc.

- *Special linear algebra tools.* The computation of the basis extension, i.e., the REF extension, can be implemented in a specialized algorithm rather than standard or sparse Gaußian elimination (e.g. see [42]). During the elimination some parts of the coefficient matrix may already be very dense. Therefore a good strategy should divide the matrix into sparse/dense blocks and deal with them separately by using specialized libraries.

- *Skip useless reductions.* The current version of the algorithm may be adapted to some analogues of Buchberger criteria for the Gröbner basis algorithm (e.g., see [93]). Some attempts for the BBA and the BBBA can be found in [59].

In the rest of the section, we outline the main idea how to skip useless reductions using the signature mechanism described in [59]. Let $V = (f_1, \ldots, f_s) \in \mathbb{B}_n^s$ be the input Boolean polynomials for the BBBA. A **signature bound** (according to [59]) for a polynomial $g \in \mathbb{B}_n$ is a pair $(i, t) \in \mathbb{N} \times \mathbb{S}^n$ which stores the history of the creation of g during the run of the BBBA. The signature bound indicates that g is some linear LT_σ-reduction of tf_i. The signature bound helps us to skip duplicate reductions coming from the symmetry. The main idea is motivated by the following proposition. However, the situation in the BBBA is more complicated because we use some polynomials in reductions whose support may not be contained in the current universe.

Proposition 3.31. *Let V be a linearly LT_σ-interreduced set of Boolean polynomials, and let $f_k, g, g', g'' \in \mathbb{B}_n$ such that $x_i x_j f_k \xrightarrow{V} g'$ and $x_j x_i f_k \xrightarrow{V} g''$. In other words, g' and g'' have the signature bounds equal to $(k, x_i x_j)$. Then $g' \xrightarrow{V} 0$ if only if $g'' \xrightarrow{V} 0$.*

Proof. Let us take a look at $x_i x_j f_k \xrightarrow{V} g' \xrightarrow{V} 0$. By Proposition 3.19 we get $x_j x_i f_k = x_i x_j f_k \in \langle V \rangle_{\mathbb{F}_2}$. We know that $g'' = x_j x_i f_k + v$ for some $v \in V$. Hence we have $g'' \in \langle V \rangle_{\mathbb{F}_2}$. From Proposition 3.19 follows that $g'' \xrightarrow{V} 0$. The opposite direction is analogous. □

The signature bounds can be implemented as a C++ `vector` S of unordered sets. The set $S[i]$ contains all terms t of the signature bounds of shape (i, t). The operations regarding the signature bounds require divisibility in $\mathbb{S}^n$ and a fast `find` method for given signature bound in S. Therefore we use `std::unordered_map::find`, because it has constant complexity in the average case.

3.8 Experiments

All timings in this section were obtained on a machine having a 2.6 GHz Intel(R) Core(TM) i7-5600U CPU and 16 GB RAM. The C++ programs were compiled using the GCC compiler version 5.3.1 with the `-O2` optimization flag. Timings that exceed the timeout limit of 1500 seconds are marked by ">1500". When measuring the time consumption, we take only the actual runtime in account. In particular, the initial memory allocation and the setup of Boolean rings are excluded. Since we are comparing algorithms called from other software systems to our native C++ implementation, that is usually faster in the initialization phase, this should be fair enough. The actual tests have been performed many times to assure that there is no disturbance during the computation which affects the running time.

To illustrate the difference between dense and sparse representations, we measure some execution times of the sparse implementation (i.e., the sparse representations of terms and coefficient matrices) of the Boolean BBA and the dense representation implemented in C++ in Table 3.1. For benchmarking, we use systems of quadratic polynomial equations coming from the algebraic attacks at Small Scale AES (ssAES, cf. [46]). The number of rounds is denoted by r, the number of rows of the state by a, the number of columns of the state by b and the size of the word by e. To these parameters we add the number of indeterminates and the number of equations.

These timings show a first Boolean BBA implementation based only on standard C++ libraries (such as `std::bitset`, `boost::dynamic_bitset`, and `std::vector`), basic profiling and a cache-friendly design. Recall that 0/1 coefficients of terms are stored in `std::bitset` and `boost::dynamic_bitset` only as one bit (plus some bytes in a header) in the memory, in contradiction to `bool`, which is usually stored in one byte, because it must be addressable. In fact, this size is platform dependent and can be larger.

From Table 3.1 one can clearly see that sparse BBBA outperforms dense the BBBA. On one hand, dense representation turns out be very effective when performing operations such as `XOR` of two rows of a matrix. On the other hand, accessing a coefficient is

Table 3.1: Timings of the implementation of the Boolean BBA in C++

Small scale AES						BBBA (dense)	BBBA (sparse)
# var	# eq	r	a	b	e	in seconds	in seconds
20	36	1	1	1	4	0.04	0.01
36	60	1	1	2	4	1.60	0.14
36	68	2	1	1	4	1.17	0.27
40	72	1	2	1	4	12.76	0.21
52	100	3	1	1	4	27.15	7.03
64	112	2	1	2	4	240.23	35.03
68	132	4	1	1	4	953.44	17.94
72	120	1	2	2	4	422.61	4.56
72	136	2	2	1	4	>1500	299.72
84	164	5	1	1	4	>1500	148.92
100	196	6	1	1	4	>1500	439.10
116	228	7	1	1	4	>1500	1045.49

expensive, because the bits are packed in the memory and therefore masking is required. Overall, sparse representation takes advantage of sparse encoding of algebraic attacks on ssAES, and in this context sparse linear algebra clearly outperforms dense techniques. Note that time performance does not depend only on the size of the input system, i.e., the number of polynomials and the number of indeterminates, but the shape of the system.

In Table 3.2 we compare our implementation of the BBBA with Gröbner basis (GB) methods. Note that implementations of Buchberger's algorithm have been carefully improved for several decades: Buchberger's criteria allow us to avoid the computation of many unnecessary S-polynomials, special data structures for Boolean polynomials have been implemented, additional criteria for avoiding unnecessary critical pairs in characteristic 2 have been developed, and so on. However, we have not found any really satisfactory implementation of the BBBA. For instance, the current implementation in `ApCoCoA` (cf. [109]) is able to do only the very smallest of the examples below in reasonable time (under 30 min.).

Hence it is quite remarkable that the preliminary implementations of the algorithms and optimizations presented in this chapter are sometimes already in the range of general Gröbner basis implementations. We expect that the next round of optimizations, including the prediction of zero reductions, i.e., the full analogues of Buchberger's criteria, will close the gap even more.

Table 3.2 gives some indications of the relative timings of various implementations to compute the border bases or Gröbner bases of some ssAES ideals. By `GBasis5` we refer to the command for the Gröbner basis computation in `CoCoA5` (see [1]), called from within the `ApCoCoA` server. By `slimgb` we refer to the command in `Singular`, called from within `Sage` [110].

These timings are still far away from the ones given in [26] for a specialized version of

Table 3.2: BBBA/GB Timings for Small Scale AES Ideals

Small Scale AES	BBA (sparse)	GBasis5	slimgb
ssAES-1-1-1-4	0.01	0.004	0.23
ssAES-1-1-2-4	0.14	0.03	0.48
ssAES-2-1-1-4	0.27	0.54	0.54
ssAES-1-2-1-4	0.21	0.06	0.60
ssAES-3-1-1-4	7.03	0.20	0.86
ssAES-2-1-2-4	35.03	3.10	1.99
ssAES-4-1-1-4	17.94	0.94	1.22
ssAES-1-2-2-4	4.56	4.39	2.11
ssAES-2-2-1-4	299.72	14.92	2.62
ssAES-5-1-1-4	148.92	2.37	1.54
ssAES-6-1-1-4	439.10	9.44	2.02
ssAES-7-1-1-4	1045.49	210.88	2.39

the `PoliBoRi` package. As we noted above, this may be partially due to the particular data structures (based on ZDDs) of this package, or to its highly optimized criteria for avoiding critical pairs.

Let us finish this section with some experiments regarding the Signature-based Boolean Border Basis Algorithm (SBBBA, see [59]) and other improvements which were briefly discussed in Sect. 3.7. In Table 3.3 we measure the total number of calls to the reduction function `Reduce` (i.e. Algorithm 3.7) in Algorithm 3.10. The SBBBA is implemented on top of the standard BBBA. The main new addition is the signature mechanism which allows us to skip some polynomials in V. The skipped polynomials are not to interreduced. For the SBBBA, we provide the total number of times when Algorithm 3.7 is applied (# red), the same signature bound occurs (# dupl), and how many polynomials in V reduce to the zero (# zero). These numbers tell us how successful the signatures are for detecting unnecessary reductions – instead of doing some reductions we just skip the polynomial. The relation between skipped signatures and the runtime of the algorithm is nonlinear, because the polynomials in the final iteration of the BBA are usually very dense, whence their reduction takes more time than the reduction of the polynomials in the very first iterations. Thus skipping such dense polynomials provides a significant speed-up.

Furthermore, note that the summation of the three rightmost columns in Table 3.3 does not give the value in the third column, because the choice of the next signature bounds guide the computation in a different way than in the standard version of the Boolean BBA.

In Table 3.4 we compare our implementation of Algorithm 3.10 and the SBBBA to `PolyBoRi`, a very good implementation of Buchberger's Algorithm for Boolean polynomials, and to the SAT solver `CryptoMiniSat` version 5 (`CMS`). For the Boolean BBA, we have in fact three versions: the standard version (see Algorithm 3.10), the signature based version, and a version of the SBBBA with substitutions. The later version shares

Table 3.3: The number of reductions in the Standard BBBA and the Signature Based BBBA

Boolean system		Standard BBBA	Signature Based BBBA		
# vars	# eq	# red	# red	# dupl	# zero
20	36	3077	1647	183	530
36	67	17243	7748	775	2241
52	99	49809	23989	3996	10235
64	111	80564	41600	11233	20767
68	131	115252	50884	9806	24715
72	119	89325	42209	5172	13675
72	135	124834	57679	14051	28782
84	163	201419	85773	17158	44382
100	195	314388	140337	34696	77360
116	227	450060	235484	78668	151900
132	259	605857	313960	91796	186899

the core of the SBBBA. However, when a polynomial of type x_i, $x_i + 1$, $x_i + x_j$, or $x_i + x_j + 1$ is discovered as a result of some row reduction, we substitute the value of x_i in all polynomials known so far, thereby reducing the complexity of the problem by one indeterminate.

By `PolyBoRi` we refer to the `PolyBoRi` implementation of Buchberger's Algorithm called from within `Sage` version 7.5.1 (see [110]). We converted the Boolean polynomials of the input to the CNF format by calling the dense ANF to CNF conversion which is a built-in function in `Sage`. The later conversion will be discussed in detail in Chapter 4.

Table 3.4 shows that the timings of the SBBBA are approaching the speed of `PolyBoRi` for smaller sized examples. The speedup provided by the signature technique seems to be around a factor of 5. For the instances in Table 3.4, the substitution technique is apparently a significant further improvement.

Table 3.4: Comparison of the BBBA and the GBA timings

Boolean system		BBBA	SBBBA	SBBBA+sub	`PolyBoRi`	`CMS`
# vars	# gens	in sec	in sec	in sec	in sec	in sec
20	36	0.04	0.03	0.01	0.22	0.01
36	67	0.24	0.30	0.09	0.62	0.02
52	99	4.97	1.70	0.79	0.97	0.03
64	111	27.31	16.97	12.45	1.79	0.06
68	131	27.35	6.79	1.68	1.25	0.06
72	119	73.24	12.20	4.59	1.80	0.06
72	135	129.58	10.96	2.80	2.22	0.06
84	163	87.83	18.19	6.05	1.69	0.07
100	195	282.23	54.06	11.61	2.10	0.07
104	199	>1500	1130.67	383.50	8.22	0.07
116	227	541.12	113.73	81.05	2.45	0.09
132	259	1087.58	252.95	152.87	3.10	0.10
148	291	>1500	473.78	263.10	3.41	0.10

Chapter 4

Integrating Algebraic and SAT Solving

Motivation Many search problems over Boolean variables can be formulated in terms of the satisfiability of a set of clauses or the solving of a system of Boolean polynomials. There exists a great variety of algorithms coming from different areas such as commutative algebra, SAT or SMT, that can be used to tackle these instances. However, their approaches to inferring new constraints vary and seem to be complementary to each other. For instance, `XOR` constraints are handled very differently in SAT solvers and in computer algebra systems. Thus, it is natural to ask: Is it possible to create a platform that combines the power of both types of solvers? The answer is "yes", and in this chapter we provide a combination of the Boolean Border Basis Algorithm (BBBA) with a CDCL SAT solver in a portfolio-based fashion. The integration has been implemented in C++.

Related work For an integration of SAT solvers with the Gröbner basis algorithm, we refer to [114], [87] or [37]. A combination of a SAT solver with the XL algorithm can be found in [30]. Yet another approach is to modify a SAT solver such that it admits `XOR` clauses (besides a set of CNF clauses), i.e. `XORs` of literals such as $x_1 \oplus x_2 \oplus x_3 = \texttt{True}$. For various methods dealing with `XOR` clauses, we refer the reader to [13, 75] or to [106] for a concrete implementation. This chapter is based on the articles [53–55].

Structure and contents To achieve the goal described above, we have to discuss conversion methods from the conjunctive normal form (CNF) to the algebraic normal form (ANF) of a Boolean function. This motivates to define algebraic and logical representations in Sect. 4.1. Whereas the reverse conversion has been studied before (see Sect. 4.2), the CNF to ANF conversion has been achieved predominantly via a standard method described in Sect. 4.3 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 in Sect. 4.4 which is geared towards producing fewer and lower degree polynomials. In Sect. 4.5, in particular, we look for as many linear polynomials as possible in the converted system and check that our algorithm finds them. In Sect. 4.6 we list some applications of given conversion methods.

After the theory of various conversions is prepared, we are ready to describe an integration of algebraic and SAT solving in Sect. 4.7 together with some implementation remarks in Sections 4.8 and 4.9. Instead of building a complete fusion or a theory solver for a particular problem, both solvers work independently and interact through a communication interface. Hence a greater degree of flexibility is achieved. The SAT solver

`antom`, which is currently used in the integration, can be easily replaced by any other CDCL solver. Moreover, the ideas described in this chapter can be used to tailor an integration with various algebraic solvers, not only the BBBA. Altogether, this is the first open-source implementation of the BBBA and its combination with a SAT solver. Experiments in Sect 4.10 show that the ANF produced by our algorithm outperforms the standard conversion in "real life" examples originating from cryptographic attacks, and that our integration has potential to outperform the base solvers.

4.1 Preliminaries

The following definition creates a bridge between algebraic and SAT solving. From the algebraic point of view, we are interested in zeros (i.e. assignments yielding `False`) of the system. From the logic point of view, we consider models (i.e. assignments yielding `True`) of a CNF formula.

Definition 4.1. 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. that S is an **algebraic representation** of C, if and only if $\mathcal{S}(C) = \mathcal{Z}(S)$.

The following observations are immediate, but useful.

Proposition 4.2. *Let $k \in \mathbb{N}$, and let $S_i \subseteq \mathbb{B}_n$ be sets of Boolean polynomials and C_i sets of clauses in logical variables $X_1, \ldots, X_n$ for $i = 1, \ldots k$. Then the following statements hold.*

(a) $\bigcap_{i=1}^k \mathcal{Z}(S_i) = \mathcal{Z}(\bigcup_{i=1}^k S_i)$

(b) $\bigcap_{i=1}^k \mathcal{S}(C_i) = \mathcal{S}(\bigcup_{i=1}^k C_i)$

(c) Let C_i be a logical representation of S_i for $i \in \{1, \ldots k\}$. Then $\bigcup_{i=1}^k C_i$ is a logical representation of $\bigcup_{i=1}^k S_i$.

Using Proposition 2.18 and 2.12, we get the following correspondences. Every set of clauses C corresponds to a Boolean function that can be converted to a Boolean polynomial in ANF. Conversely, every polynomial in S can be converted individually to a Boolean function and thus to CNF, and then the union of all converted clauses is the logical representation of S (see Proposition 4.2). Thus such representations exist.

In general, the algebraic representation of S is not unique. However, if $\#S = 1$, the representation is unique.

Definition 4.3. Let C be a set of clauses in n logical variables. The Boolean polynomial $f \in \mathbb{B}_n$ with $\mathcal{S}(C) = \mathcal{Z}(f)$ is called the **standard algebraic representation** of C.

The polynomial f in this definition represents the unique Boolean function $\mathbb{F}_2^n \to \mathbb{F}_2$ mapping $a \mapsto 0$ if $a \in \mathcal{S}(C)$ and $a \mapsto 1$ otherwise. The polynomial is unique as well by Proposition 2.12.

Sometimes new indeterminates x_i (or logical variables X_j) are introduced during the conversion process, e.g., during the sparse conversion in the next section. In this case, we assume that the corresponding logical variables X_i (or indeterminates x_j) are defined even if they are not used in the logical formula (or in the Boolean system).

4.2 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 Boolean 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 dense linear systems. 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 [17, 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 do not have many indeterminates. Thus one can go trough all possible assignments to construct the sparse CNF. The complexity of this conversion is clearly exponential w.r.t. the number of indeterminates appearing in the polynomial.

Example 4.4. 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 $C = \big\{\{X_1, X_2\}, \{\bar{X}_1, X_2\}, \{\bar{X}_1, \bar{X}_2\}\big\}$ 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.

$\triangle$

Dense conversion methods (see [11], [63]) introduce new variables. First, 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. I.e., we convert $x_{i_1}x_{i_2}\cdots x_{i_\ell}$ by encoding $t + x_{i_1}x_{i_2}\cdots x_{i_\ell}$ to the clauses $\{X_{i_1}, \bar{T}\}, \ldots, \{X_{i_\ell}, \bar{T}\}$, $\{\bar{X}_{i_1}, \bar{X}_{i_2}, \ldots, \bar{X}_{i_\ell}, T\}$. 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 4.5. 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 $x_1 + x_2 + x_3 + x_4$ to CNF, we get the clauses

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

$\triangle$

Let $f \in \mathbb{B}_n$ be a polynomial of degree d, let $k = \#\operatorname{Supp}(f)$, and let r be the cutting number. We introduce at most k new variables for each term. Every term is encoded to at most $d + 1$ clauses. The linear `XOR` chain of length k is divided into $\lfloor \frac{k}{r} \rfloor$ linear polynomials, and every linear polynomial is then converted to at most 2^r clauses. In total, we introduce at most $k + \lfloor \frac{k}{r} \rfloor$ new variables and at most $k \cdot (d+1) + \lfloor \frac{k}{r} \rfloor \cdot 2^r$ clauses. Both conversions suffer from the problem that breaking the `XOR` structure in the ANF tends to introduce many auxiliary indeterminates or many new clauses.

4.3 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. [60]). The detailed description is given in Algorithm 4.1.

Algorithm 4.1 `StdANF` (Standard CNF to ANF Conversion)

Input: A set of clauses 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: $S := \emptyset$
2: **foreach** c **in** C **do**
3: $\quad f := 1$
4: $\quad$ **foreach** L **in** c **do**
5: $\quad\quad$ **if** $L = X_i$ is positive **then**
6: $\quad\quad\quad f := f \cdot (x_i + 1)$
7: $\quad\quad$ **else if** $L = \bar{X}_i$ is negative **then**
8: $\quad\quad\quad f := f \cdot (x_i)$
9: $\quad\quad$ **end if**
10: $\quad$ **end foreach**
11: $\quad S := S \cup \{f\}$
12: **end foreach**
13: **return** S

Proposition 4.6. *Algorithm 4.1 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 $L_i = \bar{X}_i$. The rest follows from Proposition 4.2. □

Let us apply this algorithm to a concrete case.

Example 4.7. 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.

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

△

Clearly, Algorithm 4.1 performs at most $\#C \cdot n$ multiplications in $\mathbb{B}_n$. Notice that its output f for a single input clause c is the standard algebraic representation of c. Moreover, $\deg(f)$ equals the length of the clause c in the algorithm. A clause c containing only positive literals is converted to a polynomial having $2^{\#c}$ terms in its support. Hence even a small set of clauses may be converted to a rather dense 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.

One way how to overcome this exponential blowup in the support is to introduce a new indeterminate for every negation of a logical variable. The exact procedure is given in Algorithm 4.2.

Note that Algorithm 4.2 produces only a term (with a linear polynomial) per clause. The proof of correctness follows from Proposition 4.6. Let C be a set of clauses in n logical variables with $\#C = k$ and $\ell = \max_{c \in C} \#c$. The conversion produces k monomials of degree at most ℓ and at most n linear trinomials.

4.4 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 4.8. (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** (or **negative**) **sign** if the number of negative literals is an even (or odd) number.

Algorithm 4.2 `ExtANF` (Extended Standard CNF to ANF Conversion)

Input: A set of clauses C in logical variables $X_1, \ldots, X_n$.

Output: A set $S \subseteq \mathbb{B}_{2n}$ of Boolean polynomials defined over the indeterminates $x_1, \ldots, x_n, y_1, \ldots, y_n$ such that S is an algebraic representation of C. (The corresponding logical variables Y_i are not used in C.)

1: $S := \emptyset$
2: **foreach** c **in** C **do**
3: $f := 1$
4: **foreach** L **in** c **do**
5: **if** $L = X_i$ is positive **then**
6: $f := f \cdot (y_i)$
7: $S := S \cup \{x_i + y_i + 1\}$
8: **else if** $L = \bar{X}_i$ is negative **then**
9: $f := f \cdot (x_i)$
10: **end if**
11: **end foreach**
12: $S := S \cup \{f\}$
13: **end foreach**
14: **return** S

(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 $\#\big(\operatorname{Var}(c) \cap \operatorname{Var}(c')\big) \geq m$ is called an **overlapping number** of c and c'.

Given a number m, Algorithm 4.3 decomposes a set of clauses C into blocks B_c for $c \in C$ such that m is an overlapping number of c and every clause in B_c.

Notice that some clauses in C may not be included in the set $\mathcal{B}$ produced by Algorithm 4.3. 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 4.3 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 4.9. *The output of Algorithm 4.3 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 $\#\big(\operatorname{Var}(c) \cap \operatorname{Var}(c')\big) \geq m$. Clearly, Step 2 of Algorithm 4.3 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 4.3 is uniquely determined by C and m, and does not depend on the order in which the clauses c are selected in Step 1. $\square$

Algorithm 4.3 `Blocks` (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 $\left(\bigcup_{B \in \mathcal{B}} B\right) \cup T = C$, and every clause in T contains less than m literals.

1: **foreach** c **in** C **do**
2: $\quad B_c := \left\{c' \in C \mid \#\big(\operatorname{Var}(c) \cap \operatorname{Var}(c')\big) \geq m\right\}$
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** $(\mathcal{B}, T)$

The elements of the set $\mathcal{B}$ returned by Algorithm 4.3 will be called the m**-blocks** of C. Let us apply Algorithm 4.3 to some easy cases.

Example 4.10. 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 4.11. In the setting of Example 4.7, Algorithm 4.3 calculates the following two 2-blocks.

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

△

Now we are ready to present the main Algorithm 4.4.

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 4.15), 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 4.12. *The output of Algorithm 4.4 is an algebraic representation of C and is uniquely determined by σ and m.*

Algorithm 4.4 `BlockANF` (Blockwise CNF to ANF Conversion)

Input: A set of clauses C in logical variables $X_1, \dots, X_n$, 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.
Require: Algorithm 4.1 and 4.3, a reduced Boolean Gröbner basis algorithm.

1: $S' := \emptyset$
2: $(\mathcal{B}, T) :=$ `Blocks`(C, m).
3: $\mathcal{B} := \mathcal{B} \cup \bigcup_{t \in T} \{t\}$
4: **foreach** B **in** $\mathcal{B}$ **do**
5: $\quad Q :=$ `StdANF`(B)
6: $\quad$ Let G be the reduced Boolean σ-Gröbner basis of the ideal $\langle Q \rangle$.
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** $S_{\sigma,m}$

Proof. First we prove that $S_{\sigma,m}$ is an algebraic representation of C. In Step 3 we have the equality $\bigcup_{B \in \mathcal{B}} \left(\bigcup_{c \in B} c \right) = C$, 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 4.3. We know that Q is an algebraic representation of $B \in \mathcal{B}$ in Step 5 by Proposition 4.6. Furthermore, G is an algebraic representation of B as well, because $\langle Q \rangle = \langle G \rangle$. Clearly S' is an algebraic representation of C in Step 9 by Proposition 4.2. Since LT_σ-interreduction does not change the set of zeros, we get that $S_{\sigma,m}$ is an algebraic representation of C.

By Proposition 4.9 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. $\square$

Example 4.13. Let us apply Algorithm 4.4 in the setting of Example 4.7.

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

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

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 4.7. $\triangle$

In the following we study Step 6 of Algorithm 4.4 in more detail. Note that the application of Algorithm 4.2 instead of Algorithm 4.1 would not make any difference, in principle, because forming the S-polynomial of a term $y_i t$ for some squarefree term t and a linear polynomial $y_i + x_i + 1$ is equivalent to substituting $y_i = x_i + 1$ into $y_i t$, and hence the output of Algorithm 4.1 is obtained.

Claims (a)–(d) of the following proposition 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 basic logical reasoning.

Proposition 4.14. *In the setting of Algorithm 4.4, 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 $I = \langle Q \rangle$.*

(a) 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 $\langle Q \setminus \{q_j\} \rangle$.

(b) 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 $\langle Q \setminus \{q_{i_1}, \ldots, q_{i_s}\} \rangle$.

(c) 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 $\langle Q \setminus \{q_i\} \rangle$.

(d) 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_i and c_j on the variable X_e. Then the standard algebraic representation of r can be derived from q_i, q_j by polynomial calculus.

(e) We have $\mathcal{S}(B) = \emptyset$ if only if $G = \{1\}$.

(f) 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 $\max\{\deg(g) \mid g \in G\} \leq \deg(q_j)$.

(g) 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. (a) 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.

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

(c) 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.

(d) The standard algebraic representation of c_i (or c_j) is $q_i = (x_e + 1)f$ (or $q_j = x_e g$) for some polynomials f, g. Then the algebraic representation of r is the polynomial $fg = g(x_e + 1)f + f x_e g = g q_i + f q_j$.

(e) 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 contained in the ideal. Hence the claim follows from the strong version of Hilbert's Nullstellensatz.

(f) Let s be an S-polynomial that is produced when computing G. Every term in the support of s divides $\mathrm{LT}_\sigma(q_j)$. Hence the claim follows.

(g) 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 of $\mathrm{LT}_\sigma(f) = \mathrm{LT}_\sigma(g)$, 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 cannot be reduced further. Thus f is contained in some reduced Boolean σ-Gröbner basis of the ideal I. Now the claim follows form the uniqueness of the reduced Boolean σ-Gröbner basis. □

Notice that Claim (g) can also be found in [22, Thm. 5.3.5.]. Moreover, the algebraic representation of r in Claim (d) is an S-polynomial if w and w' do not share any variable. Claim (d) implies that resolution can be efficiently simulated by polynomial calculus when Algorithm 4.2 is used for the conversion. In more detail, let $\ell = \max\{\#c_i, \#c_j\}$. Then we perform at most $\ell(\ell-1)$ multiplications of terms to obtain gq_i (or fq_j) from q_i (or q_j) and one addition to get fg.

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

Definition 4.15. 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 of length ℓ consists of $2^{\ell-1}$ 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 4.9) having only positive, or only negative clauses of length ℓ as nodes.

Proposition 4.16. *Let K be a complete signed set of clauses with positive (or negative) sign and $\mathrm{Var}(K) = \{X_{i_1}, \dots, X_{i_\ell}\}$. Then $x_{i_1} + \cdots + x_{i_\ell} + 1$ (or $x_{i_1} + \cdots + x_{i_\ell}$) 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 $\mathrm{Var}(K)$. 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 4.5 illustrates the previous proposition. A lower number of clauses can also produce linear polynomials, but we have to allow clauses of different lengths.

Proposition 4.17. *(a) Let φ, ψ be propositional logic formulas in n logical variables. Then φ is equivalent to the formula $\varphi' = (\varphi \vee \psi) \wedge (\varphi \vee \bar{\psi})$, i.e. we have $\mathcal{S}(\varphi) = \mathcal{S}(\varphi')$.*

(b) Let c, w be clauses. The set $C = \{c\}$ is equivalent to $C' = \{c \cup w, c \cup \bar{w}\}$, i.e. we have $\mathcal{S}(C) = \mathcal{S}(C')$.

(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 $c \cup \{L_1^*, \ldots, L_k^*\}$, where L_i^* equals to L_i or $\bar{L}_i$.*

Proof. Claim (a) follows easily by comparing the truth tables of 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 4.18. 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 4.4 converts B to $x_1 + x_2 + x_3$ and $x_2x_3 + x_2 + x_3 + 1$. △

Proposition 4.17 leads to the following combinatorial test for checking whether the conversion of a set of clauses B contains a linear polynomial.

Proposition 4.19. *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 $j = 1, \ldots, k$, we define the following sets:*

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

(a) *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 $x_{i_1} + \cdots + x_{i_\ell} + 1$.

(b) *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 follows analogously. In view of Proposition 4.17, the set $B^+_{j,V}$ contains all possible extensions of c_j in the set of 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 $\#\big(\bigcup_{j=1}^k B^+_{j,V}\big)$ and obtain the claimed formula. □

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 4.5 is a straightforward application of Proposition 4.19. 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

Algorithm 4.5 `LinANF` (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 an algebraic representation of B contains L.

1: $L' := \emptyset$
2: **foreach** subset $V = \{X_{i_1}, \ldots, X_{i_\ell}\}$ **of** $\mathrm{Var}(B)$ **do**
3: $B^+ := \emptyset$
4: $B^- := \emptyset$
5: **for** $j = 1, \ldots, k$ **do**
6: $B^+ := B^+ \cup B^+_{j,V}$
7: $B^- := B^- \cup B^-_{j,V}$
8: **end for**
9: **if** $\#B^+ = 2^{\ell-1}$ **then**
10: $L' := L' \cup \{x_{i_1} + \cdots + x_{i_\ell} + 1\}$
11: **end if**
12: **if** $\#B^- = 2^{\ell-1}$ **then**
13: $L' := L' \cup \{x_{i_1} + \cdots + x_{i_\ell}\}$
14: **end if**
15: **end foreach**
16: Let L be an LT_σ-interreduced $\mathbb{F}_2$-basis of $\langle L' \rangle_{\mathbb{F}_2}$.
17: **return** L

according to sign. Note that the ideas behind Algorithm 4.5 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 [106].)

Since we have to check all subsets of $\mathrm{Var}(B)$ in Step 2, Algorithm 4.5 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 4.5, the following proposition shows that Algorithm 4.4 produces at least the same number of linear polynomials.

Proposition 4.20. *Let C be a set of clauses, let $I \neq \langle 1 \rangle$ be the ideal generated by an algebraic representation of C, and let σ be a degree compatible term ordering.*

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

(b) Let $m = 1$ be an overlapping number. Let L be the output of applying Algorithm 4.5 to C. Let S be the output of applying Algorithm 4.4 to (C, σ, m). Then we have $\#L \leq \#\{s \in S \mid \deg(s) = 1\}$.

Proof. (a) Let $f \in I = \langle G \rangle$ be a linear polynomial in L. 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 are done, we still have a linear polynomial in G.

(b) Assume that Algorithm 4.5, applied to the input C, has discovered a set of clauses K which contains a complete signed block after extension. Using Algorithm 4.3 on the input (C,m), we compute a pair $(\mathcal{B},T)$. If $\#\operatorname{Var}(K) = 1$, then the corresponding linear polynomial is derived from a clause in T, and we are done. If $\#\operatorname{Var}(K) \geq 2$, then K appears in a block $B \in \mathcal{B}$, and we can use (a). □

4.6 Some Applications of the Conversion Algorithms

In this section we collect some applications of the conversion methods discussed above, and in particular, of Algorithm 4.4.

Identifying `XORs` Algorithm 4.3 builds blocks of clauses that tend to contain a complete signed sets of clauses. The reduced Boolean Gröbner basis discovers the linear polynomials corresponding to such blocks (cf. Proposition 4.20). Hence Algorithm 4.4 can be used to find `XOR` constraints from a CNF formula as in Algorithm 4.5. These constraints may be then processed by a linear algebra solver.

Solving by conversions During our experiments, we found that the conversion of a Boolean system S 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 ssAES-1-1-1-4 and ssAES-2-1-1-4. These polynomials can be generated in `Sage` [110]. 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 4.6.

Algorithm 4.6 `GenLin` (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 $\langle S \rangle$.
Require: Algorithm 4.4, a sparse conversion method described in Sect. 4.2.

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

Algebraic solving of CNF Converting a simple Boolean system S to CNF using the sparse strategy and back to polynomials by Algorithm 4.1, gives us usually a rather

denser and higher-degree system $\hat{S}$. Even in the cases when the computation of a Gröbner basis of the ideal $\langle S \rangle$ is done in seconds, a Gröbner basis of the ideal $\langle \hat{S} \rangle$ may take hours to compute. Algorithm 4.4 provides more suitable ANFs for the computation of a Gröbner basis of the ideal $\langle S \rangle$. This improvement is made explicit and quantified in Section 4.10.

However, computing a Gröbner basis of $\langle S \rangle$ is still not feasible for larger instances. Note that, for SAT solvers, input CNFs defined using 10000 variables are usually toy instances. On the other hand, polynomial systems using 10000 indeterminates are most likely beyond the reach of algebraic solvers. To give a sense of scale, the Gröbner basis routine implemented in [20] did not finish under 3000 seconds for any ANF instance listed in Table 4.1. Thus applying algebraic solvers to the conversion of a set of clauses is promising only if we integrate powerful logical reasoning (such as resolution) into them. First steps in this direction are done in the next section.

Hybrid reasoning Conversions from CNF to ANF give us another perspective on SAT solving, and vice versa. In the next section we look closer at resolution, which is the most fundamental rule of inference in SAT solvers, from the algebraic point of view. This approach enables to define a new data structure and algebraic rules of inference which generalize and improve resolution in Chapter 5.

Hybrid solving When SAT solvers target really hard problems, an integration with an algebraic solver may come in handy. Algorithm 4.4 can be used to form a basic skeleton for integrating algebraic and SAT solvers. The SAT solver sends a “new” set of clauses C to the algebraic solver. Then Steps 1–8 are executed in a similar fashion as in Algorithm 4.4. (The Gröbner basis algorithm can be substituted by any other algebraic method.) Instead of appending the result G, the algebraic solver sends `StdANF`(G) back to the SAT solver. There is a great degree of flexibility in this approach, e.g., special filtration methods can be applied to choose which clauses resp. polynomials are selected for transmission. The details are given in the next section.

4.7 The Integration of the BBBA with a SAT Solver

Many search problems can be encoded as systems of Boolean polynomials or SAT-instances. That means that one can solve the same problem with algebraic means such as the Boolean Border Basis Algorithm (BBBA) or a SAT solver individually. The conversion methods discussed in this chapter transform a Boolean system to a CNF formula (and vice versa) such that the $\mathbb{F}_2$-rational zeros of the system correspond to the satisfying assignments of the logical formula. Thus we may run both solvers in parallel and let them interchange the “new information”, or one solver can dynamically help the another one with a certain subproblem, etc. We will focus on the scenario when an algebraic solver helps a SAT solver because it provides the best results according to our initial experiments.

Previously, we had handled the interaction of two solvers manually. During our experiments, several examples were observed where one solver is sped up by utilizing

information derived by the other. Based on these observations, the communication was automated with a view towards optimizing the achievable gains.

The integration is tailored to be applicable for most CDCL SAT solvers. For our experiments, we used the SAT solver `antom` [101]. Modern SAT solvers are mainly based on CDCL. They produce many *learned clauses* which contain new information that can be potentially used in the BBBA after a conversion. Conversely, any new polynomial found in the ideal by the BBBA can be converted and sent to a SAT solver. To reduce the amount of information that needs to be transferred, we transmit only short clauses and short polynomials of a low degree. Moreover, an additional filtering technique has been developed to further reduce the number of clauses that are handled by the BBBA. This selection strategy makes the BBBA sufficiently fast to keep up with the SAT solver. In this way the BBBA is not stuck with computations which are potentially outdated and irrelevant for the SAT solver by the time they are finished.

Description of the integration. Assume that the CDCL SAT solver is running on a given CNF (interpreted as a set of clauses W) in the background. Our approach is divided into 7 steps (viewed from the BBBA side) which repeat until the SAT solver stops. The diagram of the integration is depicted in Figure 4.1. Note that Algorithm 3.10 can be replaced by any other algebraic reasoning in general, and the filtering is scaled by four parameters $a, b, c, d \in \mathbb{N}$.

(1) *Receiving clauses.* The SAT solver sends a set of new learned clauses C with $\#C = a$ that has generated to the BBBA.

(2) *Clause filtration.* We define a subset $C' \subseteq C$, where C' contains clauses $k \in C$ such that there exist $k' \in C$ with $k \neq k'$ that shares at least one variable with k. We buffer only the first b clauses on an as-they-come basis.

(3) *Converting clauses to polynomials.* We use the standard conversion in Algorithm 4.1 to produce a set of Boolean polynomials S from the selected clauses.

(4) *Computing a border basis.* We call Algorithm 3.10 on the ouput of the previous step. We restrict the sets of indeterminates of the Boolean ring to the indeterminates actually appearing in the input polynomials. We do not apply `FinalReduction`. Call the result S'.

(5) *Polynomial filtration.* We consider only the polynomials in $S' \setminus S$ of degree $\leq c$. Among them, we select polynomials with the smallest support. Let Q be the result.

(6) *Converting polynomials to clauses.* We convert these polynomials to clauses via the (sparse) truth-table method described in Example 4.4. We buffer only the first d clauses on an as-they-come basis. Call the result K.

(7) *Sending the clauses.* We send these clauses to the SAT solver and go to Step (1).

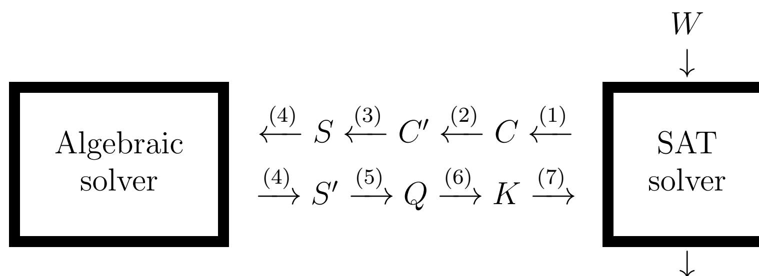


Figure 4.1: The integration of an algebraic solver with a SAT solver.

First of all, let us prove that the integration is correct.

Proposition 4.21. *The above integration of the solvers is correct.*

Proof. Let c be a clause sended in Step (7). We have to show that $\mathcal{S}(W \cup \{c\}) = \mathcal{S}(W)$. The inclusion "$\subseteq$" is trivial. Let us prove the inclusion "$\supseteq$" by contradiction.

Suppose that there exists $a \in \mathcal{S}(W)$ such that $a \notin \mathcal{S}(W \cup \{c\})$. That means $a \notin \mathcal{S}(\{c\})$. There exists a polynomial $q \in Q$ in Step (5) such that $\mathcal{Z}(\{q\}) = \mathcal{S}(\{c\})$. Backtracking the steps to Step (1) will get

$$a \notin \mathcal{Z}(\{q\}) \supseteq \mathcal{Z}(Q) \supseteq \mathcal{Z}(S') = \mathcal{Z}(S) \supseteq \mathcal{S}(C') \supseteq \mathcal{S}(C) \supseteq \mathcal{S}(W).$$

The last inclusion follows form the fact that the conflict clauses are logical consequence of the input formula W. The equality $\mathcal{Z}(S) = \mathcal{Z}(S')$ follows from $\langle S \rangle = \langle S' \rangle$. Hence we get $a \notin \mathcal{S}(W)$, which is the contradiction. □

The integration can be used to find a model of an arbitrary CNF formula in the DIMACS format (i.e., the standard format for SAT solvers). This option seems to be effective when the input contains a rich algebraic structure, e.g., many `XOR` constraints. Note that such benchmarks are quite common in cryptanalysis. Most SAT solvers do not use the rich `XOR` structure hidden in modern cryptosystems (e.g., in ARX ciphers or in permutation-substitution networks, etc.) at all. Thus an integration with the BBBA, which naturally works with addition modulo 2, may come in handy.

In the following toy examples we illustrate the synergy of the BBBA with a resolution-based SAT solver.

Example 4.22. Let $f = 1 + x_2 + x_1x_2$ and $g = 1 + x_1 + x_1x_2$ both from $\mathbb{B}_2$. It is easy to verify that there exists no common $\mathbb{F}_2$-rational zero of $\{f, g\}$. Let us compute $f+g = x_1+x_2$ in order to get rid of the leading term. We convert f (and $f+g$) into CNF via the sparse method described in Example 4.4 and get $\big\{\{X_1, X_2\}, \{\bar{X}_1, X_2\}, \{\bar{X}_1, \bar{X}_2\}\big\}$ $\big($and $\big\{\{X_1, \bar{X}_2\}, \{\bar{X}_1, X_2\}\big\}\big)$. Resolution then yields $\{X_1\}$ and $\{\bar{X}_1\}$, and hence certifies inconsistency. Note that the BBBA can find the element 1 as follows. Firstly, multiply $f + g$ by x_1 and get $x_1 + x_1x_2$. Secondly, interreduce the latter polynomial with g and get 1. △

Let us see how the BBA can profit from intermediate results of a SAT solver. Assume that a SAT solver finds out an assignment for one indeterminate. For the BBBA, this translates to a new polynomial of the form $x_i + a_i$, where $a_i \in \mathbb{F}_2$. Normally, the BBBA will not use this information very efficiently, as the following example shows.

Example 4.23. Suppose that we are computing the Boolean border basis of an ideal in $\mathbb{B}_4$ and that the SAT solver provides us with the insight that $x_1 = 0$. Moreover, assume that we are currently trying to simplify the polynomial $x_1x_2x_3x_4 + x_1x_2x_3 + x_3$. In Algorithm 3.10 we have to wait for the result of two $V^{(+)}$ computations to eliminate $x_1x_2x_3$, and then one more to eliminate $x_1x_2x_3x_4$. However, a direct substitution of $x_1 \mapsto 0$ immediately shows $x_3 = 0$, and by substituting $x_3 \mapsto 0$, we can simplify other polynomials. △

Therefore, whenever the SAT solver discovers useful constraints, such as $x_i + a_i = 0$ or $x_i + x_j = 0$, it is better to perform the corresponding substitutions everywhere in the BBBA and thus reduce the number of variables involved in the system. Next we sketch two cases that are inconvenient for a SAT solver, but can be easier for Algorithm 3.10.

Firstly, if V contains two Boolean polynomials v_i and v_j having the same leading terms and such that $\#(\mathrm{Supp}(v_i) \cap \mathrm{Supp}(v_j))$ is large enough, the Boolean polynomial $v_i + v_j$ is much simpler. In other words, addition in the BBBA helps us to derive new polynomials efficiently. Because a standard SAT solver does not support the `XOR` operation, it will tend to discover the clauses corresponding to $v_i + v_j$ much later. Secondly, multiplying by an indeterminate while computing $V^{(+)}$ may lead to lower degree and simpler polynomials. For instance, by multiplying $x_1x_2x_3 + x_2x_3 + x_1x_3 + x_1x_2 + x_2 + x_3 + 1$ by x_1 we get only x_1. In a SAT solver, this new information may not be discovered so quickly.

4.8 Design of the Communication

To combine the power of the SAT solver with the advanced reasoning of the BBBA, a severe communication challenge has to be overcome. While it would be possible to create a fully integrated BBBA-SAT hybrid solver, the maintenance of such a solver would be difficult and the implementation of new features into either base solver challenging. Therefore, a communication framework that allows the exchange of data between the border basis and the SAT solver is developed instead. The design of this communication layer is focused on two objectives.

(a) The overhead for the data transfer must be low.

(b) The base solvers should be modified as little as possible.

To achieve the first design goal, a shared memory communication approach is chosen. By defining a shared memory region that is accessible to both solvers, large amounts of data can be transmitted at extremely high speeds. To satisfy the second objective, the communication is restricted to consist only of clauses. Furthermore, the shared memory communication is implemented with the help of the `Boost Interprocess Library` [47]. This library allows different processes to access a common, shared memory region. Thus,

the SAT solver and the BBBA can be executed independently and simply access the same shared memory region.

The communication itself is combined into a handler class that performs the generation of the shared memory region and the coordination and synchronization of the data access. It furthermore provides a simple interface for the sending and receiving of clauses. The handler class only needs to be instantiated by the solvers to gain access to the shared memory region.

When the BBBA and the SAT solver are combined, there are two instances of the shared memory manager that are communicating. Apart from the initialization of the shared memory region that is performed at the start of the application, these instances behave exactly in the same way. For simplicity, the two managers are referred to as m_1 and m_2 here. To allow for an efficient communication, each manager utilizes its own shared memory area for outgoing clauses, o_1 and o_2. This enables a fast full duplex communication, as each manager can send and receive at the same time.

When m_1 is asked to transmit a clause to m_2, it first stores the clause in a local queue. The next clause of the local queue is transferred to the shared region o_1 when m_2 indicates that it read the previously shared clause from that area. Once the clause has been written to o_1, m_1 informs m_2 that a new clause is available. The manager m_2 then copies the clause from o_1 to its own memory region and marks the clause as read. Thus, the next clause in the queue of m_1 can be transmitted.

To avoid any idle waiting in the background, the check for new clauses and the transmission of the next clause are only performed when the solvers update their shared memory handler.

4.9 Modifications of the SAT Solver

The SAT solver constantly generates new conflict clauses. The shear volume of conflict clauses makes it unfeasible to share all of them with the BBBA. Instead, only conflict clauses below a certain size threshold are transmitted. A new conflict clause is transmitted immediately after it has been generated. This modification adds only a single line of code to the solver.

Receiving new clauses is slightly more challenging because of the way clauses are stored and considered in the solver `antom` [101]. For efficiency reasons, the first literal of every clause that is not satisfied must be free (i.e., currently not assigned to a value). Therefore, each new clause that is received from the BBBA is first sorted and then added. Depending on the variable assignment that is currently under consideration by the SAT solver, a new clause might, furthermore, be unsatisfied at the moment. In this case, the solver backtracks until the clause is not unsatisfied anymore. Here the new clause acts similar to a conflict clause and guides the solver away from unsatisfied regions of the search space. The additional tasks that are required to handle a received clause are placed into a new function. Thus, the main SAT solver code only needs to be extended by a single line of code that checks for the arrival of new clauses.

The shared memory handler is updated once after every decision. This is sufficiently often to receive any new clauses, but does not add any undue overhead to the solver.

Overall, the SAT solver is modified only to a very small extent. Hence the solver can be freely developed without worrying about complex dependencies. Similarly, it should be comparatively easy to add the presented communication layer to a different SAT solver, should the need arise.

4.10 Experiments

In this section we examine the efficiency of the several methods for the CNF to ANF conversion and an integration of the BBBA with a SAT solver. All tests were executed on a computing server having a 3.00 GHz Intel(R) Xeon(R) CPU E5-2623 v3 and a total of 48 GB RAM. The block-conversion algorithm was prototypically implemented in `python` 2.7 using the `PolyBoRi` library [17] integrated in `Sage` [110]. The timeout for testing various conversions was set to 1500 seconds.

In order to compare conversion methods, we choose specific mid-size instances from the following benchmark suites: the logical representations of the encryption of the Small-scale AES cipher (ssAES) [46] and factoring of integers [14]. Instances of type ssAES-a-b-c-d represent propositional formulae in CNF derived from the gate level circuit implementation of Small-scale AES with a rounds, a 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 two given primes a, b. Instances of type ssAES-`2-2-4-fa` represent algebraic fault attacks on ssAES-10-2-2-4 with different faults injected (see [28]).

Table 4.1 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 4.1 and 4.4 together with execution times. We use $\sigma =$ `DegRevLex` and $m = 2$. The latter parameter performs the best, because it does not create big blocks B_c in Algorithm 4.3 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 4.1 we clearly see that the algebraic representation given by Algorithm 4.4 produces lower degree polynomials than the one by Algorithm 4.1. While Algorithm 4.1 usually produces only very few linear polynomials, the table shows that Algorithm 4.4 tends to return enough linear polynomials to eliminate approximately one third of all indeterminates. Moreover, we note that Algorithm 4.4 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 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 $\prod_{i=1}^{\ell}(x_i + a_i)$ 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.

Table 4.1: Number of converted polynomials by degree

Instance	CNF		Algorithm 4.1				Algorithm 4.4			
	#vars	#cl.	#l.	#q.	#h.	sec	#l.	#q.	#h.	sec
ssAES-2-1-2-4	237	701	1	360	340	0.01	70	453	0	2.16
ssAES-2-1-4-4	412	1218	1	598	619	0.02	132	746	0	5.43
ssAES-2-2-2-4	526	1615	1	716	898	0.05	201	882	0	8.56
ssAES-2-4-1-4	669	2065	1	960	1104	0.06	241	1190	0	13.55
ssAES-2-2-4-4	935	2883	1	1196	1686	0.08	375	1491	0	23.92
ssAES-10-1-2-4	1081	3361	1	1792	1568	0.09	337	2194	1	40.22
ssAES-2-4-2-4	1157	3652	1	1434	2217	0.11	501	1778	0	36.16
ssAES-10-1-4-4	1862	5824	1	2986	2837	0.15	604	3692	0	106.17
ssAES-2-4-4-4	2077	6596	1	2394	4201	0.25	957	2978	0	108.70
ssAES-10-2-2-4	2441	7841	1	3584	4256	0.24	947	4406	0	173.31
ssAES-10-2-4-4	4289	13904	1	5986	7917	0.57	1785	7353	0	521.28
ssAES-10-4-1-4	3149	10065	1	4800	5264	0.33	1149	5913	0	308.90
ssAES-2-1-2-8	4664	13974	1	9102	4871	0.59	343	12971	0	603.53
fact-151-283	271	1333	2	250	1081	0.02	154	361	2	3.62
fact-59-1009	328	1640	2	294	1344	0.03	184	447	2	5.31
fact-373-929	328	1640	2	294	1344	0.03	170	487	2	5.28
fact-1777-491	403	2029	2	354	1673	0.05	202	611	2	7.94
fact-2393-3371	466	2380	2	400	1978	0.07	224	730	2	10.72
fact-9601-10067	638	3296	2	532	2762	0.06	288	1056	2	19.63
fact-12601-18701	745	3853	2	616	3235	0.08	341	1226	2	27.17
fact-59441-62201	826	4312	2	676	3634	0.09	365	1395	2	33.33
fact-81551-100057	947	4945	2	770	4173	0.14	407	1633	2	45.08
fact-583909-600203	1280	6784	2	1010	5772	0.18	527	2271	2	104.36
ssAES-2-2-4-fa	1310	4510	73	1952	2485	0.13	694	2704	197	78.31
ssAES-2-2-4-fa	1310	4510	73	1952	2485	0.12	694	2705	196	77.89
ssAES-2-2-4-fa	1310	4510	73	1952	2485	0.12	694	2707	194	77.52
ssAES-2-2-4-fa	1310	4510	73	1952	2485	0.10	738	2515	182	77.38
ssAES-2-2-4-fa	1310	4510	73	1952	2485	0.10	696	2704	195	77.81
ssAES-2-2-4-fa	1310	4510	73	1952	2485	0.10	696	2700	199	77.51
ssAES-2-2-4-fa	1310	4510	73	1952	2485	0.13	731	2555	180	77.45
ssAES-2-2-4-fa	1310	4510	73	1952	2485	0.13	696	2702	197	77.28

Proposition 4.14 states that Algorithm 4.4 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 4.4 on C together with the newly found clauses (e.g., conflict clauses). We believe that this would produce even more low-degree polynomials.

To evaluate the combination of the BBBA and `antom` implemented in C++, two different kinds of experiments have been performed. We note here that `antom` is deterministic, i.e. it gives the same result on the same input for each run.

The first set of experiments is meant to showcase the general usefulness of the information that is derived by the BBBA for the SAT solver. Let C be an input set of clauses for `antom`. We convert C into a set of Boolean polynomials S via the standard conversion. Next we run the BBBA for 5 minutes and then stop the execution. We select one linear polynomial f from its output manually and convert f back to CNF via the sparse method. Let C' be the resulting set of clauses. We run `antom` twice: once with the input C and then with the input $C \cup C'$. The timings in Table 4.2 illustrates

the speed-up obtained by manual section of extra information provided by the BBBA.

Table 4.2: Comparison of timings of `antom` on the Small Scale AES instances in [46] without vs with extra clauses corresponding to a linear polynomial.

`CNF instances`	`antom`	`antom` + lin. poly
ssAES-2-1-2-8	11.80	3.50
ssAES-2-2-1-8	111.59	88.15
ssAES-2-2-4-4	196.06	21.76
ssAES-1-2-4-8	666.93	209.11
ssAES-2-4-2-4	3997.91	1432.24

For the second set of experiments, Algorithm 3.10 and the integration framework with `antom` in C++ as described in Section 4.7 were used. In Table 4.3 we present the timings of this automation on various benchmarks. Instances `fact`-x-y were generated by [14]. They encode the factoring problem for $x \cdot y$. The other benchmarks encode algebraic attacks or algebraic fault attacks on the cryptosystems Small Scale AES (ssAES) and LED-64. For the full description of these benchmarks, we refer to [28, 46]. The timeout limit was set to 2500 seconds.

During our experiments we found examples where the integration was slower than the SAT solver by itself. In practice, we therefore suggest running the SAT solver alone on one machine and the integration in parallel on another machine. In this way, we cannot be "unlucky" and we will always profit from the best timings. This is particularly relevant in cryptanalytic scenarios where the solution of an instance implies breaking a cryptosystem.

Notice that the timings of the integration of the two solvers are sometimes not stable, i.e. two timings for the same instance may differ substantially. These differences occur because new clauses are added at different points in time.

Let us discuss the positive effect of a new clause coming from algebraic reasoning on a SAT solver. There are these basic scenarios.

(a) The new clause is not satisfied by the partial model constructed by the SAT solver, and hence the SAT solver is forced to backtrack.

(b) The new clause eliminates a large portion of assignments, and hence these assignments do not have to be considered.

(c) The new clause effects the statistics carried out by the SAT solver. That is why a different decision is chosen in the next level of branching by the heuristics.

On one hand, scenarios (a) and (b) have a positive effect on solving. On the other hand, scenario (c) may affect the integration in both positive and negative ways, depending on the state of the SAT solver and the example under consideration.

We have extended our experiments to the SAT competition benchmarks (see [8, 9] for the benchmark descriptions). The results are shown in Figure 4.2. We choose

Table 4.3: Timings of the integration of the BBBA with `antom` vs vanilla `antom` for various SAT instances

`CNF instances`	`antom`	BBBA + `antom`
`fact-81551-100057`	0.23	0.22
`ssAES-2-2-4faultInNibble1with1faultyBits`	3.12	2.35
`fact-3981643-3981641`	7.83	6.91
`fact-2190823-2190821`	19.53	74.25
`fact-7367627-7367621`	29.18	146.13
`fact-12619463-12619427`	40.43	101.34
`ssAES-4-4-4faultInNibble1with4faultyBits`	41.15	55.87
`LED-64faultInNibble1with1faultyBits`	45.70	55.38
`ssAES-4-4-4faultInNibble1with1faultyBits`	49.02	48.61
`fact-5160011-5160007`	63.98	55.47
`fact-5621809-5621809`	81.54	110.07
`fact-4752977-4752949`	207.18	189.58
`fact-5308571-5308553`	282.91	37.22
`ssAES-2-2-4-4algebraicCNF`	268.70	235.39
`fact-49987277-49999553`	337.58	45.29
`fact-12598967-12598951`	441.88	78.60
`fact-4593761-4593737`	527.22	10.11
`fact-5287813-5287801`	605.76	102.48
`fact-5620907-5620907`	653.63	5.04
`fact-10000079-10000019`	> 1200	760.41

the parameters $(a, b, c, d) = (40, 10, 3, 20)$. On the x-axis we put the names of the CNF instances, and on the y-axis the CPU timings are displayed. We focused on the examples where the integration outperforms the vanilla `antom`. On one hand, there exist cases where the integration is slower. On the other hand, we found 8 instances, for which the integration finished before the timeout, whereas the vanilla `antom` did not. (These cases are not displayed in the figure.)

The filtration techniques described in Section 4.7, as well as the integration itself, are still preliminary. Our next goal in the next chapter is to develop deeper understanding of the synergy of both solvers. The main difficulty is that SAT solvers use various heuristics for literal assignment and for the choice of the clause to work on next. This makes it very hard to analyze which extra clauses from the BBBA affect the timings most. Nonetheless, our results show that the additional information from the BBBA may already greatly increase the speed of the SAT solver.

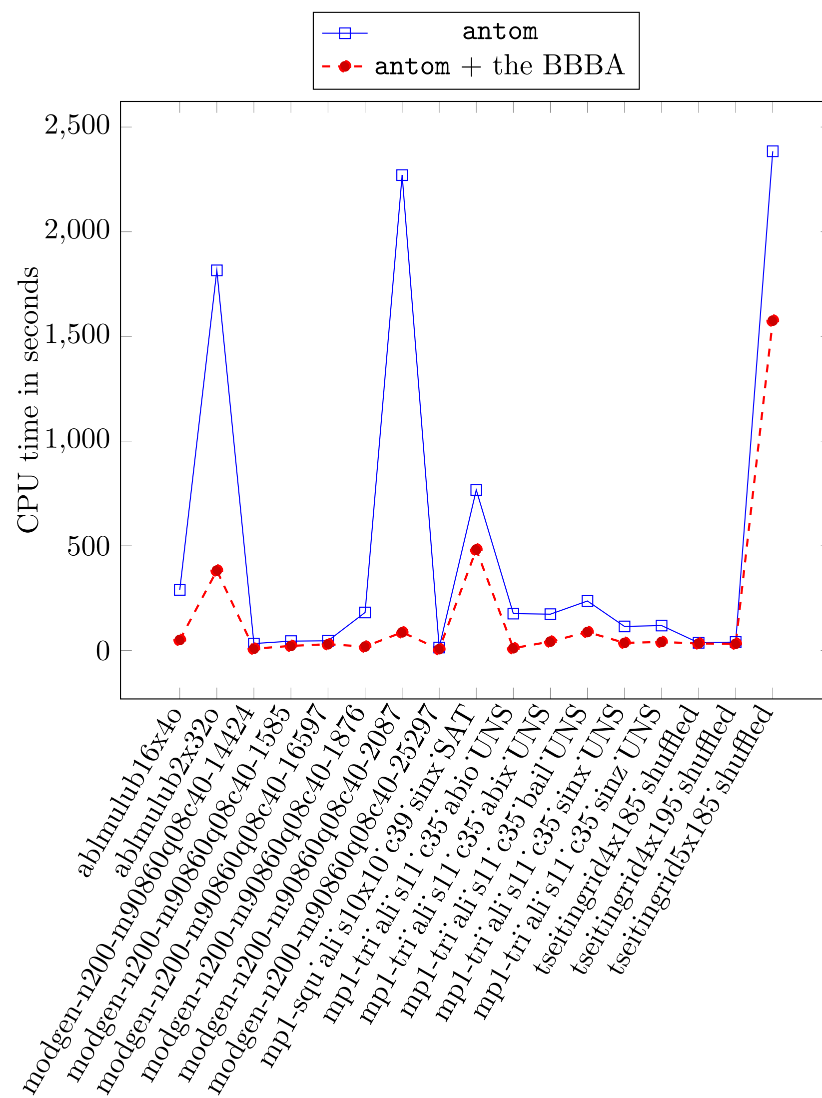


Figure 4.2: Measuring CPU timings of `antom` and the `antom` integration with the BBBA tested on various CNF benchmarks

Chapter 5

Proof Systems and SRES

Motivation In this chapter, we look into abstract concepts how algebraic or logic solvers derive a solution of a search problem, or how they certify unsatisfiability. On one hand, a solver can find a single solution of an instance "by luck", e.g. a SAT solver (depending on a branching heuristic) chooses the "right" order of guesses. On the other hand, in the case of an unsatisfiable problem, a solver has to systematically certify that all assignments do not fit the initial constraints, e.g. by deriving a semantic implication that is clearly unsatisfiable. Thus we focus on certifying the unsatisfiability because it is very often more difficult.

Solvers use various rules of inference how to obtain "new information" and work with different syntaxes that determine their data structures. A proof system is a theoretical concept that helps us understand the capability of the underlying syntax and its rules of inference.

Because individual proof systems may have some weaknesses, it is reasonable to combine them into stronger ones. In this chapter we study a combination of resolution and polynomial calculus. In Sect. 4.3 we remarked that a clause $c = \{L_1, \ldots, L_m\}$ with literals L_i corresponds to the polynomial $f = \ell_1 \cdots \ell_m$ where $\ell_i = x_i + 1$ for $L_i = X_i$ and $\ell_i = x_i$ for $L_i = \bar{X}_i$. We generalize this concept, and we allow $\ell_1, \ldots, \ell_m$ to be arbitrary linear polynomials over $\mathbb{F}_2$. By translating a linear polynomial ℓ_i back to logic, L_i becomes a XOR clause, and the product f becomes a linear clause.

Using this terminology, that we later call linearly split polynomials, we taylor a special rule of inference, called s-resolution (sres), and a new proof system SRES with the following properties.

- The rule sres generalizes the classical resolution known from logic (see Definition 2.24). More precisely, resolution corresponds to s-resolution with $s = 1$ when clauses are converted by Algorithm 4.1.
- The rule sres mimics the construction of S-polynomials (see Definition 2.22) for polynomials that are products of linear polynomials, i.e. sres is taylored to cancel the leading terms.
- The rule sres (together with other rules in SRES) allows Gaußian elimination from linear algebra to be incorporated in the proofs.

Because of its strong algebraic background and the above facts, we call sres an algebraic extension of resolution. Our main motivation for this chapter is to lift up the following concepts known from SAT solving of sets of clauses to solving of set of linear clauses.

- *Syntax:* clauses $\rightarrow$ linear clauses.

- *Transformations:* resolution $\to$ s-resolution.
- *Proof systems:* the resolution proof system RES $\to$ the combined proof system SRES.
- *Closure algorithms:* the resolution closure algorithm $\to$ the SRES closure algorithm.
- *Constraint propagation:* Boolean constraint propagation $\to$ Gaußian constraint propagation.
- *Solvers:* DPLL-based SAT solvers using RES $\to$ DPLL-based solvers using SRES.

Related work Let us mention some previous contributions to combining the resolution calculus and the polynomial calculus into a new proof system, and compare them to s-resolution. The proof system RES-LIN in [61] admits linear clauses, which correspond to our linearly split polynomials, but uses only 1-resolution steps, whereas the proof system in [13] works only for `XOR` clauses (i.e., linear Boolean polynomials) and relies on Gaußian elimination. Moreover, the proof system RLIN in [96] applies a more general addition of linear factors than s-resolution and does not target the main idea of S-polynomials, namely cancellation of leading terms. Apparently, these proof systems have not led to efficient implementations. For a recent account of the theory on proof complexity, we refer the reader to [71]. This chapter is based on [55].

Structure and contents The chapter is organized as follows. In Section 5.1 we recall basic definitions regarding proof systems, and we provide various examples of proof systems coming from logic and algebra. In Section 5.2 we define linearly split polynomials and the proof system SRES which incorporates and extends the resolution proof system. Moreover, we establish a relation with other combined systems that use resolution and polynomial calculus. In particular, we show that SRES simulates the proof system RLIN defined in [61, 96].

In Section 5.3 we prove that the SRES proof system is implicationally and refutationally complete, and in Sect. 5.4 we give a few proofs in SRES that illustrate some advantages over non-combined proof systems. In Section 5.5 we describe concrete algorithms that searches for SRES-refutation proofs of inconsistent systems based on computing closure or DPLL. In Sect. 5.7 we conclude this chapter with some experiments that compare our implementation of the above mentioned solvers to SAT solvers.

5.1 Preliminaries

Let us start the section with the definition of a proof system known from mathematical logic.

Definition 5.1. A **proof system**, sometimes called a **Hilbert system** or **Hilbert calculus**, consists of

- a syntax, i.e. a set of rules which determine the set of well-formed formulas of the system,
- a finite set of axioms, i.e. a set of formulas which are assumed to be tautologies,
- a finite set of sound rules of inference.

The previous definition is due to Hilbert and Frege. In the field of proof complexity, a *Frege proof system* (as in in [71, Ch. 2]) is defined as a polynomial time proof-verification algorithm, i.e. for all propositional formulas φ the following holds: φ is unsatisfiable if and only if there exists a string (proof) p such that the algorithm accepts the input (p, φ).

Next we recall the definition of a proof.

Definition 5.2. Let PS be a proof system. Let $F_1, \ldots, F_m, G_1, \ldots, G_k, H$ be propositional formulas.

(a) A PS-**proof** of a formula H from the **initial premises** $F_1, \ldots, F_m$ in the proof system PS is a sequence of formulas $\pi = (G_1, \ldots, G_k)$ such that $G_k = H$ and each of the formulas G_i is of one of the following forms.

  (i) $G_i \in \{F_1, \ldots, F_m\}$

  (ii) G_i is one of the axioms of PS.

  (iii) G_i is obtained from some formulas G_j with $j < i$ by applying one of the rules of inference of the proof system PS.

(b) In the setting of (a), the number k is called the **length** of the proof π, and the formula G_i is called a **line** in the proof π.

(c) In the setting of (a), if H is equal to a contradiction $\perp$, then the proof π is called a PS-**refutation** of $F_1, \ldots, F_m$.

(d) If a formula H has a PS-proof from the premises $F_1, \ldots, F_m$, we write $F_1, \ldots, F_m \vdash_{\mathsf{PS}} H$, or simply $F_1, \ldots, F_m \vdash H$ if no confusion can arise.

Sometimes, we form proofs in PS as directed acyclic graphs instead of sequences. The vertices are labeled by formulas used in the proof, and the edges $(G_{i_1}, G), \ldots, (G_{i_j}, G)$ denote the fact that G is derived from $G_{i_1}, \ldots, G_{i_j}$ by applying one of the rules of inference of PS, i.e. we have $G_{i_1}, \ldots, G_{i_j} \vdash_{\mathsf{PS}} G$. By a *tree-like proof*, we mean the proof which corresponds to a tree.

The following definition provides further useful properties of proof systems. They are motivated by the question whether any semantic implication is provable in a given proof system.

Definition 5.3. (a) A proof system PS is called **implicationally complete** if for every formula H and every set of formulas $\{F_1, \ldots, F_m\}$ such that we have $F_1, \ldots, F_m \models H$, we have $F_1, \ldots, F_m \vdash_{\mathsf{PS}} H$.

(b) A proof system PS is called **refutationally complete** if for every inconsistent set of formulas $\{F_1, \ldots, F_m\}$, i.e. for $F_1, \ldots, F_m \models \bot$, we have $F_1, \ldots, F_m \vdash_{\mathsf{PS}} \bot$, where the symbol "$\bot$" denotes a contradiction.

In the language of proof complexity, Definition 5.1 is similar to *Frege systems* with two important modifications. Firstly, a Frege system works with formulas built in an arbitrary way by a (functionally complete) set of connectives, e.g. not only with CNFs in the case of the connectives "$\wedge$" and "$\vee$". Secondly, a Frege system is always implicationally complete.

When refuting a Boolean formula, we ask two fundamental questions.

Problem:	Existence and computability of proofs
Input:	A Boolean formula φ, a proof system PS that admits φ.
Question 1:	Is there a "short, narrow" PS-refutation of φ?
Question 2:	If the answer to Question 1 is positive, find a "short, narrow" PS-refutation of φ.

Without going too much into details, by "short" proofs we mean proofs that have polynomial length w.r.t. the size of the input formula. By "narrow" proofs we mean proofs that have polynomial size of their lines w.r.t. the size of the input formula. For instance, if for certain class of formulas there are no "short" proofs in PS, there is no way how to design practical search algorithms for these inputs using the rules of inference of PS. Conversely, if such a "short" proof exists, there is a chance to construct effective search algorithms. That is why theoretical results in proof complexity are very important for practical solving methods.

In the following we recall some basic proof systems from logic and algebra. We start with logic first.

Definition 5.4. The **resolution proof system** RES is defined as follows.

- The proof system RES admits sets of clauses.
- (No axiom schemata.)
- Its rules of inference is

$$\frac{c \cup \{X_i\} \qquad c' \cup \{\bar{X}_i\}}{c \cup c'} \quad (\mathsf{res})$$

  where c, c' are clauses and X_i is a logical variable such that neither X_i nor $\bar{X}_i$ are contained in $c \cup c'$.

The next propositions collect some useful facts about the system RES.

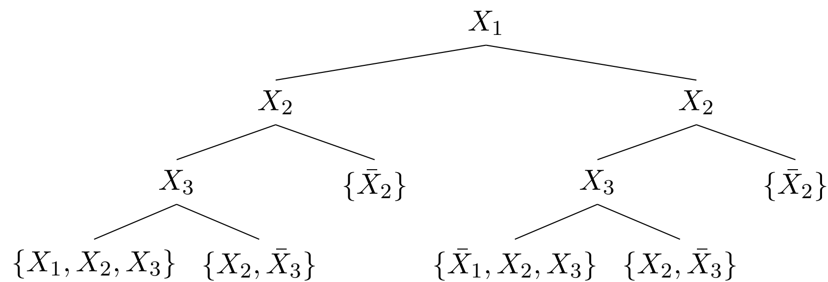
Proposition 5.5. *(a) The proof system* RES *is refutationally complete.*

(b) The proof system RES *is not implicationally complete.*

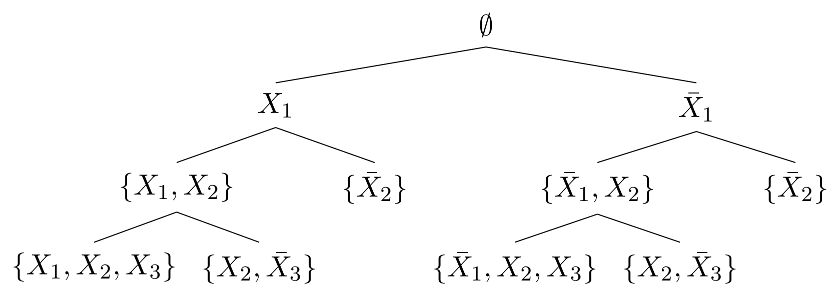
(c) *Assume that the algorithm* DPLL *(described in Algorithm 2.6) returns* False *on a set of clauses C. Then there exists a tree-like* RES*-refutation of C whose length is at most the number of decisions in the run of* DPLL.

Claim (b) is easy to prove by a counterexample. Let $c' = \{X_1, X_2\}$ and $c = \{X_1\}$. Then $c \models c'$, but c' can not be derived by resolution. Notice that Claim (c) implies Claim (a) because DPLL returns False if and only if $\mathcal{S}(C) = \emptyset$. Instead of giving a complete proof of Claim (c), we illustrate how to construct a RES-refutation from a DPLL decision tree when BCP (i.e. Algorithm 2.4) is not used. The full proof can be found in [100, Prop. 2.3]. Claim (a) is also proven in [27, Thm. 4.1.5].

Example 5.6. Let $C = \big\{\{X_1, X_2, X_3\}, \{\bar{X}_1, X_2, X_3\}, \{X_2, \bar{X}_3\}, \{\bar{X}_2\}\big\}$. The following decision tree illustrates a run of DPLL (without BCP) on C. Its nodes are branching variables. The edge going left (or right) from the node X means assigning $X =$ False (or $X =$ True). The leaves contain a clause in C that is not satisfied by the assignment along the path going to that leave.



The resolution refutation goes then backwards from leaves to the root of the tree. The resolving variable is equal to the branching variable. The completed resolution tree is given below.



Notice that BCP and multiple occurrences of the same literal in a clause can be handled similarly by the rule res as well. △

The connection between DPLL and RES is indeed very interesting. Firstly, DPLL is just tree-like resolution by Proposition 5.5(c). Thus tree-like resolution is not very efficient, e.g. each derived clause is used only once, which means DPLL has to derive the same clause every time it is used. In contradiction, in [100, Ch. 4] it is shown that CDCL

with restarts corresponds to general resolution. However, there exist CNF formulae such that any RES-refutation requires exponentially many lines.

One canonical class of such formulas are so-called *pigeonhole formulas* which encode the problem of placing n pigeons into $n+1$ holes such that there is no hole with more than one pigeon. Equivalently, the problem states the fact that there is no injection $\{1,\ldots,n+1\} \to \{1,\ldots,n\}$. Another class of examples which are hard for resolution are so-called *Tseitin formulas*. Tseitin formulas encode inconsistent linear systems based on the task of finding vertex coloring of connected graphs. For further details on hard formulas w.r.t. resolution, see [76].

Next we enhance the system RES with an extension rule. It allows new proofs which can be viewed as shortcuts in RES-proofs.

Definition 5.7. The proof system ERES is defined as follows.

- The proof system ERES shares the same syntax with RES.
- (No axiom schemata.)
- Its rules of inference are res and the extension rule def (with no premises) defined as follows.

$$\frac{\quad}{C'} \quad (\mathsf{def})$$

  where C' is a set of clauses that encodes the equivalence $Y \Leftrightarrow F$ for an arbitrary formula F and a fresh logical variable Y, i.e. Y does not appear in F (or in the previous definitions or in the input formula).

A ERES-proof can be divided into two stages. Firstly, the input set of clauses C is extended to $C^* \supseteq C$ by applying def, and then the proof from C^* is derived as in RES. Tseitin proposed to use $C' = \big\{\{\bar{Y}, \bar{X}_1, \bar{X}_2\}, \{Y, X_1\}, \{Y, X_2\}\big\}$ to encode the formula $Y \Leftrightarrow \bar{X}_1 \vee \bar{X}_2$ (e.g., see [74]). That means that we are allowed to add definitions of the form $Y := \bar{X}_1 \vee \bar{X}_2$. Even this simple definition makes the resulting system ERES very powerful, e.g. classes of formulas that would require very "long" ERES-proofs are not known. Unfortunately, there is no practical SAT algorithm based on ERES yet. Some attempts in this direction can be found in [5].

Next we discuss algebraic proof systems.

Definition 5.8. Polynomial calculus PC, sometimes called the **Gröbner proof system**, is defined as follows.

- The proof system PC admits polynomials f where $f \in \mathbb{F}_2[x_1,\ldots,x_n]$.
- Its axioms are the **Boolean axioms** $x_i^2 + x_i$ for $i = 1,\ldots,n$.

- Its rules of inference are

$$\frac{f \quad g}{f+g} \ (\mathsf{pca}) \qquad \text{and} \qquad \frac{f}{x_i f} \ (\mathsf{pcm})$$

  where $f, g \in \mathbb{F}_2[x_1, \ldots, x_n]$ and x_i is an indeterminate.

Note that we have to adjust Definition 2.17 for the algebraic setting, i.e. we say that polynomials $f_1, \ldots, f_m \in \mathbb{F}_2[x_1, \ldots, x_n]$ semantically imply a polynomial $g \in \mathbb{F}_2[x_1, \ldots, x_n]$ if $\mathcal{Z}(f_1, \ldots, f_m) \subseteq \mathcal{Z}(g)$. In this case we write $f_1, \ldots, f_m \models g$. Some useful facts about PC are given in the next proposition.

Proposition 5.9. *Let $F = \langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$.*

(a) PC *is implicationally complete.*

(b) PC *is refutationally complete.*

(c) Let S be a set of polynomials in $\mathbb{F}_2[x_1, \ldots, x_n]$. Let σ be a term ordering. A reduced σ-Gröbner basis of $\langle S \cup F \rangle$ is equal to $\{1\}$ if and only if $\mathcal{Z}(S) = \emptyset$. Moreover, there exists a PC*-proof for any polynomial in the output of Algorithm 2.1 executed on the input S.*

Proof. Let us prove Claim (a). Let $f_1, \ldots, f_m, g \in \mathbb{F}_2[x_1, \ldots, x_n]$ such that $f_1, \ldots, f_m \models g$. From Proposition 2.14 it follows that $g \in \langle f_1, \ldots, f_m \rangle$, i.e. g can be derived from pca and pcm starting from $f_1, \ldots, f_m$. Claim (b) follows from Claim (a). The proof of Claim (c) follows from [22, Sect. 2.2.]. □

A classical way how to use the rules of inference of PC for a set of clauses is via the conversions described in Algorithms 4.1 and 4.2. In the case of using Algorithm 4.2, we call the resulting proof system PCR (polynomial calculus resolution).

Definition 5.10. Let $P = \mathbb{F}_2[x_1, \ldots, x_n]$. A **Nullstellensatz proof** of a polynomial $h \in P$ from polynomials $f_1, \ldots, f_m \in P$ is a tuple of polynomials $\pi = (p_1, \ldots, p_m, r_1, \ldots, r_n) \in P^{m+n}$ such that the equation

$$\sum_{i=1}^{m} p_i f_i + \sum_{j=1}^{n} r_j (x_j^2 + x_j) = h$$

holds in P. The **degree** of the proof π is defined as

$$\max \big\{ \max_i \big(\deg(p_i) + \deg(f_i) \big), \max_j \big(\deg(r_j) + 2 \big) \big\}.$$

5.2 An Algebraic Extension of Resolution

In this section we consider Boolean polynomials which are products of linear polynomials. For instance, such polynomials emerge naturally from Algorithm 4.1. We generalize the concept to products of arbitrary linear polynomials (i.e. not only of the form x_i or $x_i + 1$). Algebraic proof systems such as PC work with polynomials that are in the expanded form. However, expanding long products of polynomials is very costly.

Using Algorithm 4.3, we can identify blocks that correspond to linear polynomials. Instead of reducing the blocks as in Algorithm 4.4, we can substitute the linear polynomials into the remaining polynomials to get a compact encoding as illustrated in the next example.

Example 5.11. Consider the set of clauses $C = \{c_1, \dots, c_7\}$ with $c_1 = \{\bar{X}_1, X_2, X_3\}$, $c_2 = \{X_1, \bar{X}_2, X_3\}$, $c_3 = \{X_1, X_2, \bar{X}_3\}$, $c_4 = \{\bar{X}_1, \bar{X}_2, \bar{X}_3\}$, $c_5 = \{\bar{X}_4, X_5\}$, $c_6 = \{X_4, \bar{X}_5\}$, $c_7 = \{X_1, X_4\}$. Using Algorithm 4.3 with $m = 2$, the 2-blocks $B_1 = \{c_1, c_2, c_3, c_4\}$, $B_2 = \{c_5, c_6\}$, and $B_3 = \{c_7\}$ are constructed. The block B_1 (or the block B_2) corresponds to $x_1 + x_2 + x_3$ (or $x_4 + x_5$). The clause c_7 has the standard algebraic representation $(x_1 + 1)(x_4 + 1)$. The final encoding of C is given by the single polynomial $(x_2 + x_3 + 1)(x_5 + 1)$ and by two relations for the "free" indeterminates $x_1 = x_2 + x_3$ and $x_4 = x_5$. △

More precisely, our main data structure is the following representation of products of linear Boolean polynomials.

Definition 5.12. Let $\mathbb{L}_n$ be the set of all linear polynomials in $\mathbb{B}_n$, i.e., the set of all polynomials of degree ≤ 1. (Here we use $\deg(0) = -1$.)

(a) For a Boolean polynomial $h \in \mathbb{B}_n$, we say that h **splits linearly**, or that h is a **linearly split polynomial**, if $h = \ell_1 \cdots \ell_k$ with linear polynomials $\ell_i \in \mathbb{L}_n$ such that ℓ_i is not divisible by ℓ_j and such that $\ell_i \neq \ell_j + 1$ for $i, j \in \{1, \dots, k\}$ with $i \neq j$.

(b) The set of all Boolean polynomials in $\mathbb{B}_n$ which splits linearly is denoted by $\mathbb{S}^n$.

(c) For $h \in \mathbb{S}^n$ which splits linearly into $h = \ell_1 \cdots \ell_k$, the set $H = \{\ell_1, \dots, \ell_k\}$ is called the **set of linear factors** of h.

(d) In the setting of (c), the number $\#H = \deg(h)$ is also called the **degree** of H.

(e) For $\ell \in \mathbb{L}_n$, we let $\mathrm{Var}(\ell)$ be the set of indeterminates occurring in ℓ. In the setting of (c), we then call $\mathrm{size}(H) = \#\,\mathrm{Var}(\ell_1) + \cdots + \#\,\mathrm{Var}(\ell_k)$ the **size** of H.

Notice that we do not count the number of terms in the support of an expanded linearly split polynomial, since we never multiply the factors out. An arbitrary product of linear Boolean polynomials can be reduced to a linearly split one, since we have $\ell^2 = \ell$ and $\ell(\ell + 1) = 0$ for $\ell \in \mathbb{L}_n$. For instance, $h = x_1^2 \cdot 1$ simplifies to $h = x_1$.

In the next definition, we define a linear clause which corresponds to a clause where literals are substituted by linear `XORs`.

Definition 5.13. (a) A Boolean formula $L_{1,1} \oplus \cdots \oplus L_{1,n_1}$ with literals $L_{i,j}$ is called a `XOR` **clause**.

(b) A Boolean formula which is a disjunction of `XOR` clauses, i.e. which is of form $(L_{1,1} \oplus \cdots \oplus L_{1,n_1}) \vee \cdots \vee (L_{1,1} \oplus \cdots \oplus L_{1,n_k})$ with literals $L_{i,j}$, is called a **linear clause**.

Throughout this chapter we consider the obvious correspondences between the following types of objects, where $\ell_1, \ldots, \ell_k \in \mathbb{L}_n$.

(C1) A linearly split polynomial $h = \prod_{i=1}^{k} \ell_i$ with $\ell_i \in \mathbb{L}_n$.

(C2) A set of linear polynomials $H = \{\ell_1, \ldots, \ell_k\} \subseteq \mathbb{L}_n$.

(C3) A linear clause $(\ell_1+1)\vee\cdots\vee(\ell_k+1)$, where the linear polynomials ℓ_i are interpreted as `XOR` clauses.

To simplify the notation, we view the sets in (C2) as polynomials in (C1), e.g., we speak of zeros of H instead of zeros of h. From the propositional logic point of view, a linearly split polynomial thus corresponds to a disjunction of a `XOR` of literals. The notion of linear clauses has been considered for instance in [61, 96].

Next we give examples of the above correspondence.

Example 5.14. (a) Let $H = \{x_1 + x_2 + 1,\, x_1 + x_3\} \subseteq \mathbb{L}_3$. Then $h = (x_1 + x_2 + 1)(x_1 + x_3) = x_1x_3 + x_1x_2 + x_2x_3 + x_3$, and H (or h) corresponds to the propositional logic formula

$$\varphi = (X_1 \oplus X_2) \vee (X_1 \oplus X_3 \oplus 1).$$

(b) In $\mathbb{B}_4$, consider the linearly split polynomial $h = (x_1 + 1)(x_2)(x_3 + x_4 + 1)$. Clearly, this polynomial is an algebraic representation of the propositional logical formula $\varphi = X_1 \vee \bar{X}_2 \vee (X_3 \oplus X_4)$. △

Furthermore, we observe that two different subsets in (C2) may represent the same polynomial, as the following example shows.

Example 5.15. Consider the sets of linear polynomials $\{x_3,\, x_2 + x_3,\, x_1 + x_3 + 1\}$ and $\{x_3,\, x_1 + x_2,\, x_1 + x_3 + 1\}$ as in (C2). Then we have $x_3(x_2 + x_3)(x_1 + x_3 + 1) = x_3(x_1 + x_2)(x_1 + x_3 + 1)$ in $\mathbb{B}_3$, i.e. the corresponding polynomials in (C1) agree. △

Next we define a new version of the classical resolution rule for linearly split polynomials. Notice that the derived polynomial remains inside the set $\mathbb{S}^n$ of products of linear Boolean polynomials.

Definition 5.16. Let $s \in \mathbb{N}_+$, let $\ell_1, \ldots, \ell_s \in \mathbb{L}_n$, and let $G, \tilde{G} \subseteq \mathbb{L}_n$ be the sets of linear factors of $g, \tilde{g} \in \mathbb{S}^n$, respectively. We assume that $\ell_i, \ell_i + 1 \notin G \cup \tilde{G}$ for $i = 1, \ldots, s$.

(a) The rule of inference sres defined by

$$\frac{\bigcup_{i=1}^{s}\{\ell_i\} \cup G \qquad \bigcup_{i=1}^{s}\{\ell_i + 1\} \cup \tilde{G}}{\bigcup_{i=1}^{s-1}\{\ell_i + \ell_{i+1} + 1\} \cup G \cup \tilde{G}} \quad (\mathsf{sres})$$

is called s-**resolution**. (For $s = 1$, we let $\bigcup_{i=1}^{s-1}\{\ell_i + \ell_{i+1} + 1\} = \emptyset$.)

(b) In the setting of (a), the derived set of polynomials is called the s-**resolvent** of $\bigcup_{i=1}^{s}\{\ell_i\} \cup G$ and $\bigcup_{i=1}^{s}\{\ell_i+1\} \cup \tilde{G}$.

(c) Let $f = \prod_{i=1}^{s} \ell_i$ and $\tilde{f} = \prod_{i=1}^{s}(\ell_i+1)$. Then the linearly split polynomial $h = \operatorname{lcm}(g,\tilde{g}) \cdot \prod_{i=1}^{s-1}(\ell_i+\ell_i+1)$ is called an s-**resolvent** of fg and $\tilde{f}\tilde{g}$.

For $s \geq 3$, the s-resolution rule depends on the numbering of the linear polynomials. 2-resolvents are unique because there is only one way how to form the linear polynomial $\ell_1+\ell_2+1$. However, considered as a Boolean polynomial, the s-resolvent is uniquely determined. The next example is a case in point.

Example 5.17. Resolving two sets $F = \{x_1+1,\, x_1+x_3,\, x_1+x_2+1,\, x_2+x_3+1\}$ and $G = \{x_1,\, x_1+x_3+1,\, x_1+x_2,\, x_2+x_3\}$ with the numbering $\ell_1 = x_1+1$, $\ell_2 = x_1+x_3$, $\ell_3 = x_1+x_2+1$, $\ell_4 = x_2+x_3+1$ yields the 4-resolvent $R_1 = \{x_3,\, x_2+x_3,\, x_1+x_3+1\}$. If we swap the last two polynomials in G, 4-resolution with the numbering $\ell_1 = x_1+1$, $\ell_2 = x_1+x_3$, $\ell_4 = x_1+x_2+1$, $\ell_3 = x_2+x_3+1$ yields $R_2 = \{x_3,\, x_1+x_2,\, x_1+x_3+1\}$. Both R_1 and R_2 correspond to the same Boolean polynomial, as we saw in Example 5.15. △

The smaller degree is obtained by a rule of inference, the better. Note that we take the union of $s-1$ elements in Definition 5.16(a). Thus the result can shrink depending on s and $G \cup \tilde{G}$. Clearly, the set $G \cup \tilde{G}$ is the set of linear factors of $\operatorname{lcm}(g,\tilde{g})$. The s-resolution inference rule can be justified as follows.

Proposition 5.18. *In the setting of Definition 5.16, let* $f = \prod_{i=1}^{s} \ell_i$ *and* $\tilde{f} = \prod_{i=1}^{s}(\ell_i+1)$.

(a) We have $\prod_{i=1}^{s}(\ell_i) + \prod_{i=1}^{s}(\ell_i+1) = \prod_{i=1}^{s-1}(\ell_i+\ell_{i+1}+1)$.

(b) The set $\bigcup_{i=1}^{s-1}\{\ell_i+\ell_{i+1}+1\} \cup G \cup \tilde{G}$ *is the set of linear factors of* $(f+\tilde{f}) \cdot \operatorname{lcm}(g,\tilde{g})$.

Proof. Let us prove Claim (a) by induction on s. The equality clearly holds for $s=1$, where the empty product is 1, as usual. Assuming the equality to hold for s, we now prove it for $s+1$. We have

$$E \;=\; \prod_{i=1}^{s}(\ell_i+\ell_{i+1}+1) \;=\; \left(\prod_{i=1}^{s-1}(\ell_i+\ell_{i+1}+1)\right) \cdot (\ell_s+\ell_{s+1}+1).$$

The induction hypothesis yields $E = \left(\prod_{i=1}^{s}(\ell_i) + \prod_{i=1}^{s}(\ell_i+1)\right) \cdot (\ell_s+\ell_{s+1}+1)$. Now we multiply the right-hand side and use $\ell_s^2 = \ell_s$ and $\ell_s(\ell_s+1) = 0$ in $\mathbb{B}_n$. We obtain

$$\begin{aligned} E \;&=\; \ell_s \prod_{i=1}^{s}(\ell_i) + \ell_s \prod_{i=1}^{s}(\ell_i+1) + \ell_{s+1} \prod_{i=1}^{s}(\ell_i) + \ell_{s+1} \prod_{i=1}^{s}(\ell_i+1) + \prod_{i=1}^{s}(\ell_i) + \prod_{i=1}^{s}(\ell_i+1) \\ &=\; 2 \cdot \prod_{i=1}^{s}(\ell_i) + \prod_{i=1}^{s+1}(\ell_i) + \ell_{s+1} \prod_{i=1}^{s}(\ell_i+1) + \prod_{i=1}^{s}(\ell_i+1) \\ &=\; \prod_{i=1}^{s+1}(\ell_i) + (\ell_{s+1}+1) \prod_{i=1}^{s}(\ell_i+1). \end{aligned}$$

This completes the proof of (a). Claim (b) follows immediately from (a). □

The next corollary shows that we can think of 1-resolution as an ordinary resolution rule known from logic.

Corollary 5.19. *Let $s \in \mathbb{N}_+$, and let $h \in \mathbb{B}_n$ be an s-resolvent of fg and $\tilde{f}\tilde{g}$ as in Definition 5.16(c).*

(a) The s-resolution rule is sound, i.e., we have $\mathcal{Z}(h) \supseteq \mathcal{Z}(fg) \cap \mathcal{Z}(\tilde{f}\tilde{g})$.

(b) Let $s = 1$, and let $C, \tilde{C}$, and D be the sets of clauses which are the standard algebraic representations given by Algorithm 4.1 of $fg, \tilde{f}\tilde{g}$ and h, respectively. Then D is the resolvent of C and $\tilde{C}$ in the usual sense of propositional logic.

Proof. To show (a), we note that, by part (b) of Proposition 5.18, we have $h = \mathrm{lcm}(g, \tilde{g}) \cdot (f + \tilde{f}) = \mathrm{lcm}(g, \tilde{g}) \cdot fg + \mathrm{lcm}(g, \tilde{g}) \cdot \tilde{f}\tilde{g}$ in $\mathbb{B}_n$. Hence we have $\mathcal{Z}(h) \supseteq \mathcal{Z}(\langle fg, \tilde{f}\tilde{g}\rangle) = \mathcal{Z}(fg) \cap \mathcal{Z}(\tilde{f}\tilde{g})$.

Claim (b) is an immediate consequence of the definitions. □

Notice that Proposition 5.18 shows that $(f + \tilde{f}) \cdot \mathrm{lcm}(g, \tilde{g})$ can be derived from fg and $\tilde{f}\tilde{g}$ by polynomial calculus. Thus s-resolution can be viewed a special case of polynomial calculus. The following example indicates why s-resolution with $s \geq 2$ is useful.

Example 5.20. Consider the linearly split polynomials $f = x_1x_2$ and $g = (x_1+1)(x_2+1)$ in $\mathbb{S}_2$. If we apply 1-resolution with $\ell_1 = x_1$ to f and g, we get the resolvent $h = 0$, which does not provide additional information. However, 2-resolution yields the linear polynomial $h = x_1 + x_2 + 1$. Notice that the degree of the 2-resolvent is one, whereas the degree of the original polynomials is two. △

Note that the constraint $x_1 = x_2 + 1$ in Example 5.20 can be found and used by a SAT solver in the preprocessing of the clauses $\{\bar{X}_1, \bar{X}_2\}, \{X_1, X_2\}$ by inspecting strongly connected components (see [17, Ch. 12.2]). Example 5.20 can be generalized to $\ell_1\ell_2$, $(\ell_1 + 1)(\ell_2 + 1) \vdash_{\mathsf{sres}} \ell_1 + \ell_2 + 1$ with $\ell_1, \ell_2 \in \mathbb{L}_n$. However, the constraint given by this s-resolvent is not easy to detect using a SAT solver when the Boolean polynomials $\ell_1\ell_2$ and $(\ell_1 + 1)(\ell_2 + 1)$ are encoded in CNF.

Now we are able to define a new proof system which uses s-resolution. We always assume that sets of linear polynomials do not contain any duplicities, i.e. the duplicities are erased after an application of inference rules.

Definition 5.21. The proof system SRES is defined by the following parts.

- The proof system SRES admits finite subsets of $\mathbb{L}_n$.
- The axioms are the Boolean axioms $\{x_i,\, x_i + 1\}$ for $i = 1, \dots, n$.
- The rules of inference consist of s-resolution sres (for various $s \in \mathbb{N}_+$) and the following **weakening rule** weak.

$$\frac{H}{H \cup \{\ell\}} \quad (\mathsf{weak})$$

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

Since we trivially have $H \models H \cup \{\ell\}$, the proof system SRES is correct with respect to semantic implication. Let us note that the Boolean axioms immediately imply the following remark.

Remark 5.22. By applying 2-resolution to the axioms $\{x_i, x_i + 1\}$ and $\{x_i + 1, x_i\}$, we obtain that $\{0\}$ is a tautology in the proof system SRES.

The proof system SRES is not the only one that operates with linearly split polynomials (and thus linear clauses).

Definition 5.23. The proof system RLIN is defined by the following parts.

- The proof system RLIN admits finite subsets of $\mathbb{L}_n$.
- The axioms are the Boolean axioms $\{x_i,\, x_i + 1\}$ for $i = 1, \dots, n$.
- The rules of inference consist of the weakening rule weak and the addition rule add.

$$\frac{F \cup \{\ell_1\} \qquad H \cup \{\ell_2\}}{F \cup H \cup \{\ell_1 + \ell_2\}} \quad (\mathsf{add})$$

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

We conclude the section with a few comments on the relationship between SRES and RLIN. Firstly, the rule sres is taylored in a way such that the leading terms cancel. This can be viewed as a generalization of the S-polynomials from Gröbner basis theory for linearly split polynomials. The rule add cancels $\ell_1 + \ell_2$ if and only if $\ell_1 = \ell_2 + 1$ which is the setting of our 1-resolution. Secondly, the sres rule, unlike add, is capable of inferring more than one linear factor at once (using s-resolution for $s \geq 2$).

In the following proposition we show that SRES can simulate add and some other rules of inference.

Definition 5.24. Let F, G be subsets of $\mathbb{L}_n$, and let $\ell \in \mathbb{L}_n$.

(a) The rule uc defined by

$$\frac{H \cup \{1\}}{H} \quad (\mathsf{uc})$$

is called **unit cancellation**.

(b) The rule mp defined by

$$\frac{\{\ell\} \qquad H \cup \{\ell + 1\}}{H} \quad (\mathsf{mp})$$

is called **modus ponens**.

The next proposition justificates the fact that SRES incoroporates Gaußian elimination.

Proposition 5.25. *(a) The rules* uc*,* mp *and* add *are sound.*

(b) Furthermore, any proof derived using uc*,* mp*, and* add *can be rewritten into an* SRES*-proof.*

(c) Let $H = \big\{\{\ell_1\}, \ldots, \{\ell_m\}\big\}$ *with* $\ell_i \in \mathbb{L}_n$ *such that* $\mathcal{Z}(\ell_1, \ldots, \ell_m) = \emptyset$*. Then we have* $H \vdash_{\mathsf{SRES}} \{1\}$*.*

Proof. The correctness of uc follows from the observation that multiplying a Boolean polynomial by 1 does not change its zeros. Using Remark 5.22 we apply 1-resolution to $H \cup \{1\}$ and $\{0\}$, and we get H. To prove modus ponens it suffices to use sres with $s = 1$ and $G = \emptyset$. Finally, we show the soundness of add. Using the weakening rule, we infer $F \cup \{\ell_1, \ell_2 + 1\}$ from $F \cup \{\ell_1\}$ and $H \cup \{\ell_2, \ell_1 + 1\}$ from $H \cup \{\ell_2\}$. Then, using 2-resolution, we get $F \cup H \cup \{\ell_1 + \ell_2\}$. This proves (a) and (b). Claim (c) follows from Claim (b) because it is possible to simulate Gaußian elimination on the linear system $\ell_1 = \cdots = \ell_m = 0$ over $\mathbb{F}_2$ using add. □

The SRES proof system is in fact a combined proof system in the sense of [71, Sect. 7.1]. For instance, RLIN is another example of a combined proof system which combines resolution with polynomial calculus. By Proposition 5.25, we immediately get that SRES efficiently *simulates* the system RLIN. This means that there exists a polynomial-time algorithm which translates any RLIN-proof of H from $F_1, \ldots, F_m$ to an SRES-proof of H from $F_1, \ldots, F_m$.

Notice that SRES cannot directly simulate the rule pca of the system PC because addition of two products of linear polynomials does not have to be a product of linear polynomials in $\mathbb{B}_n$, e.g., $h = x_1x_2+1$ cannot be written as a product of linear polynomials in $\mathbb{B}_3$. However, the polynomial h can be encoded in CNF and thus using linearly split polynomials.

5.3 Completeness of SRES

In this section we prove that SRES is refutationally and implicationally complete. We start with the following remark and the subsequent proposition which will become important later in the proof of Proposition 5.30.

Remark 5.26. Let $\pi = (H_1, \ldots, H_k)$ be an SRES-proof of H_k from $F_1, \ldots, F_m$ that uses only the rule sres. Let ℓ be an element of one of the sets H_i. Then ℓ lies in the $\mathbb{F}_2$-vector space generated by the linear polynomials contained in the union $\bigcup_{i=1}^{m} F_i$.

The notation introduced in the following definition will be very useful.

Definition 5.27. Let $F \subseteq \mathbb{L}_n$, and let $i \in \{1, \ldots, n\}$. Let σ be a term ordering.

(a) For $a \in \{0, 1\}$, let $F|_{x_i \mapsto a}$ denote the set which is obtained by substituting $x_i \mapsto a$ into the linear polynomials contained in F.

(b) For $\ell \in \mathbb{L}_n$ and a term ordering σ, let $F|_\ell$ denote the set which is obtained by substituting $\mathrm{LT}_\sigma(\ell) \mapsto \ell - \mathrm{LT}_\sigma(\ell)$ into the linear polynomials contained in F.

The exact term ordering used in $F|_\ell$ is not important, and hence we assume σ to be some fixed term ordering from now on.

Proposition 5.28. *Let $F \subseteq \mathbb{L}_n$ and let $\ell \in \mathbb{L}_n$. The following claims hold true for $i \in \{1, \dots, n\}$.*

(a) $F, \{x_i\} \vdash_{\mathsf{SRES}} F|_{x_i \mapsto 0}$

(b) $F|_{x_i \mapsto 0}, \{x_i\} \vdash_{\mathsf{SRES}} F$

(c) $F, \{x_i + 1\} \vdash_{\mathsf{SRES}} F|_{x_i \mapsto 1}$

(d) $F|_{x_i \mapsto 1}, \{x_i + 1\} \vdash_{\mathsf{SRES}} F$

(e) $F, \{\ell\} \vdash_{\mathsf{SRES}} F|_\ell$

(f) $F|_\ell, \{\ell\} \vdash_{\mathsf{SRES}} F$

Proof. First we prove (a). Let $\ell \in F$ be such that x_i occurs in ℓ. The polynomial $x_i + \ell$ corresponds to substituting $x_i = 0$ into ℓ. This addition can be derived in SRES using Proposition 5.25. The claims (b)–(f) follow analogously from Proposition 5.25. □

The next example illustrates this proposition.

Example 5.29. Let $F_1 = \{x_1 + x_2, x_1\}$ and $F_2 = \{x_1 + 1\}$. By Proposition 5.25, there exists an SRES-proof of the set $\{x_2 + 1, 1\}$ from F_1, F_2 which corresponds to the substitution of $x_1 = 1$ into F_1. Conversely, we can "backtrack" the substitution, i.e. we have $F_2, \{x_2 + 1, 1\} \vdash_{\mathsf{SRES}} F_1$. △

The proof of the following proposition is inspired by the corresponding proof for the classical resolution calculus (see for instance [27, Thm. 4.1.5]) and by the proof given in [96, Thm. 5.1].

Proposition 5.30. *The proof system* SRES *is implicationally and refutationally complete.*

Proof. First we show that SRES is implicationally complete. Let $F_1, \dots, F_m, H \subseteq \mathbb{L}_n$ be sets of linear polynomials such that $F_1, \dots, F_m \models H$. We want to prove that $F_1, \dots, F_m \vdash_{\mathsf{SRES}} H$. We proceed by induction on the number $k = \mathrm{size}(F_1) + \cdots + \mathrm{size}(F_m) + \mathrm{size}(H)$.

Let us consider the case $k = 0$, i.e. the case when all linear polynomials in F_i and H are constants 0 or 1. If all sets $F_1, \dots, F_m$ are equal to $\{0\}$, there is trivially an SRES-proof of $\{0\}$. All the semantic implicants of $\{0\}$ of size 0 can be derived from $\{0\}$ by the weakening rule. If there exists a set $F_i = \{1\}$, then there is trivially an SRES-proof that refutes $F_1, \dots, F_m$, and hence we can derive any linear clause by the weakening rule and

the unit cancellation rule. Finally, if $F_i = \{0, 1\}$, then F_i is simplified to $\{0\}$ by unit cancellation, and thus we can use the previous argument.

Now assume that the claim holds for some $k \geq 0$ and consider a semantic implication

$$F_1, \ldots, F_m \models H$$

in which the sum of sizes of $F_1, \ldots, F_m, H$ is at most $k + 1$. Choose $i \in \{1, \ldots, n\}$ such that x_i occurs in $F_1 \cup \cdots \cup F_m \cup H$. Given $a \in \{0, 1\}$, let $F|_{x_i \mapsto a}$ denote the set which is obtained by substituting $x_i \mapsto a$ into the linear polynomials contained in F.

By Proposition 5.28, we have $F_j, \{x_i\} \vdash_{\mathsf{SRES}} F_j|_{x_i \mapsto 0}$ for $j = 1, \ldots, m$. Moreover, we clearly have $F_1|_{x_i \mapsto 0}, \ldots, F_m|_{x_i \mapsto 0} \models H|_{x_i \mapsto 0}$. By the induction hypothesis, there is an SRES-proof of $H|_{x_i \mapsto 0}$ from $F_1|_{x_i \mapsto 0}, \ldots, F_m|_{x_i \mapsto 0}$. Altogether, we get

$$\{x_i\}, F_1, \ldots, F_m \ \vdash_{\mathsf{SRES}} \ H|_{x_i \mapsto 0}.$$

Furthermore, by Proposition 5.28, we have $H|_{x_i \mapsto 0}, \{x_i\} \vdash_{\mathsf{SRES}} H$. All in all, there is an SRES-proof π_1 of H from $\{x_i\}, F_1, \ldots, F_m$. Analogously, we get an SRES-proof π_2 of H from $\{x_i + 1\}, F_1, \ldots, F_m$.

Next we modify the derivations of π_1 (resp. π_2) such that they start from $\{x_i, x_i + 1\}, F_1, \ldots, F_m$ instead of $\{x_i\}, F_1, \ldots, F_m$ (resp. $\{x_i + 1\}, F_1, \ldots, F_m$). Note that the same rules of inference can be applied, and thus one can rewrite the proof π_1 into an SRES-proof α_1 from $\{x_i, x_i+1\}, F_1, \ldots, F_m$ of either H or $H \cup \{x_i\}$. Similarly, we rewrite π_2 into an SRES-proof α_2 from $\{x_i, x_i+1\}, F_1, \ldots, F_m$ of H or $H \cup \{x_i+1\}$. If the proof α_1 or α_2 ends with H, we are done. Otherwise, we have $H \cup \{x_i\}, H \cup \{x_i + 1\} \vdash_{\mathsf{SRES}} H$ by a single step of the 1-resolution rule, which concludes the proof. □

5.4 Some Example Proofs Using SRES

In this section we look at what kind of benefits can we get from using SRES. We shall see that some formulae are hard for a particular proof system such as RES, but have short refutations in other axiomatic systems of propositional calculus (e.g. see [113]). The following example shows an important advantage of the SRES proof system over PC. More precisely, it shows that linearly split polynomials efficiently store some dense Boolean polynomials.

Example 5.31. Consider the following subsets of $\mathbb{L}_{n+1}$

$$\begin{aligned} F_1 &= \{x_1 + x_2,\ \ldots,\ x_n + x_{n+1}\} \\ F_2 &= \{x_1 + x_2 + 1\} \\ F_3 &= \{x_2 + x_3 + 1\} \\ \vdots &= \ \vdots \\ F_{n+1} &= \{x_n + x_{n+1} + 1\} \end{aligned}$$

On one hand, it is easy to see that the system is inconsistent because substituting $F_2, \ldots, F_{n+1}$ into F_1 gives us 1. On the other hand, the input for the Gröbner basis algorithm is assumed to be expanded (i.e., not in the form of products of linear polynomials).

We can always find $n \in \mathbb{N}$ such that expanding F_1 to the polynom $f_1 = \prod_{i=1}^{n}(x_i + x_{i+1})$, which has 2^n terms, exceeds the available memory, and hence any Gröbner basis algorithm can not be applied. Notice that we have $\text{size}(F_1) = 2n$.

One workaround would be to introduce new indeterminates to break the long product, but it does not help too much because the Gröbner basis algorithm substitutes the new indeterminates back, and thus recovers the expanded polynomial F_1 again.

However, by Proposition 5.28, there exists a short SRES refutation that corresponds to the substitution of $F_2, \ldots, F_{n+1}$ into F_1. △

In the following example, taken from [32], we compare SRES to algebraic systems such as PC and the Nullstellensatz proofs.

Example 5.32. Consider the polynomials $f_1 = x_1 + x_1x_2$ and $f_2 = x_2 + x_2x_3$ in $\mathbb{F}_2[x_1, x_2, x_3]$. They encode the two implications $x_1 \to x_2$ and $x_2 \to x_3$. (E.g., if $x_1 = 1$, then x_2 is constrained to be 1.) Let us write a proof of $h = x_1 + x_1x_3$, i.e., the implication $x_1 \to x_3$, in the Gröbner proof system.

Firstly, we multiply f_2 by x_1, and we get $g_1 = x_1x_2 + x_1x_2x_3$. The addition $f_1 + g_1$ gives us $x_1 + x_1x_2x_3$. Then we compute $g_2 = x_3 \cdot f_1 = x_1x_3 + x_1x_2x_3$. Finally, we get $g_1 + g_2 = x_1 + x_1x_3$.

Note that the maximal degree appearing in the proof is 3. However, there exists a Nullstellensatz proof of the maximal degree $\mathcal{O}(\log(n))$ of $x_1 \to x_n$ from $x_1 \to x_2, \ldots, x_{n-1} \to x_n$ (see [32]). In our case, the Nullstellensatz proof is the tuple $(1+x_3, x_1, 0, 0, 0)$ because of the equality

$$(1 + x_3)(x_1 + x_1x_2) + x_1(x_2 + x_2x_3) = x_1 + x_1x_3.$$

Now we write the polynomials f_1, f_2 as $F_1 = \{x_1, 1 + x_2\}$ and $F_2 = \{x_2, 1 + x_3\}$. Resolving on x_2 yields $H = \{x_1, 1 + x_3\}$ which corresponds to the polynomial h. Note that the maximal degree in the proof is now 2, and the SRES proof is simpler than the Gröbner proof. △

Further typical examples that are easy for SRES and difficult for RES are inconsistent systems of linear equations (see [113]). By Proposition 5.25, SRES simulates the addition rule, and thus one can use Gaußian elimination to produce SRES-refutations for such systems.

Example 5.33. Consider the linear system

$$f_1 = x_1 + \cdots + x_n, \;\; f_2 = x_1 + \cdots + x_n + 1$$

over $\mathbb{F}_2$ where $n \gg 0$. On one hand, encoding f_1 and f_2 in CNF suffers from introducing either many auxiliary variables or many new clauses. On the other hand, by the weakening rule and 2-resolution we get $f_1(f_2+1)+(f_1+1)f_2 = 1$ in $\mathbb{B}_n$ immediately. △

Many CNF instances which are difficult for the resolution calculus can be naturally encoded in linearly split polynomials. One class of such examples comes from substituting linear `XORs` into literals as described in [76, Sect. 3]. Thus, in fact, linearly split polynomials naturally encode certain formulas which are very hard for the resolution calculus.

Example 5.34. Let $C = \big\{\{X_1\}, \{\bar{X}_1, X_2\}, \{\bar{X}_1, \bar{X}_2\}\big\}$ be a set of clauses in 2 logical variables. It is easy to see (e.g. by resolution) that $\mathcal{S}(C) = \emptyset$. The substitution $X_1 \mapsto \ell_1$ and $X_2 \mapsto \ell_2$ using two (homogeneous) polynomials $\ell_1, \ell_2 \in \mathbb{L}_n$ yields the unsatisfiable set of linearly split polynomials $\{\ell_1\}, \{\ell_1 + 1,\, \ell_2\}, \{\ell_1 + 1,\, \ell_2 + 1\}$. △

Since s-resolution is more powerful than 1-resolution, we expect that it discovers more linear polynomials when constructing SRES-proofs. Since Gaußian elimination can be simulated in SRES (see Proposition 5.25), we expect to find the element 1 in the ideal earlier. This intuition seems to work for some examples which are traditionally hard for 1-resolution refutation. Let us see some examples for this phenomenon.

Example 5.35. The linearly split polynomials $h_1 = x_1x_2$, $h_2 = x_1(x_3 + 1)$, $h_3 = (x_2 + 1)x_3$, $h_4 = (x_1 + 1)x_3$, $h_5 = x_2(x_3 + 1)$, and $h_6 = (x_1 + 1)(x_2 + 1)$ in $\mathbb{S}_3$ correspond to an example of a *Tseitin formula* derived from a graph (see for instance [76, Sect. 2]). Using 2-resolution, we get

$$\begin{aligned} h_1, h_6 &\vdash_{\mathsf{sres}} x_1 + x_2 + 1 \\ h_2, h_4 &\vdash_{\mathsf{sres}} x_1 + x_3 \\ h_3, h_5 &\vdash_{\mathsf{sres}} x_2 + x_3 \end{aligned}$$

Gaußian elimination then yields 1. The maximal degree (or the maximal size) reached in the proof is 2 (or 2). These values do not exceed the degree and the size of the input. △

Example 5.36. The linearly split polynomials $h_1 = x_{11}x_{12}$, $h_2 = x_{11}x_{13}$, $h_3 = x_{12}x_{13}$, $h_4 = x_{21}x_{22}$, $h_5 = x_{21}x_{23}$, $h_6 = x_{22}x_{23}$, $h_7 = (x_{11} + 1)(x_{21} + 1)$, $h_8 = (x_{12} + 1)(x_{22} + 1)$, and $h_9 = (x_{13} + 1)(x_{23} + 1)$ correspond to the *pigeonhole hole principle* in the case of 3 pigeons and 2 holes (cf. [76, Sect. 2]). Using 1-resolution, we get:

$$\begin{aligned} h_1, h_7 &\vdash_{\mathsf{sres}} x_{12}(x_{21} + 1) = r_1 \\ h_2, h_7 &\vdash_{\mathsf{sres}} x_{13}(x_{21} + 1) = r_2 \\ h_3, h_9 &\vdash_{\mathsf{sres}} x_{12}(x_{23} + 1) = r_3 \\ h_4, h_8 &\vdash_{\mathsf{sres}} x_{21}(x_{12} + 1) = r_4 \\ h_6, h_8 &\vdash_{\mathsf{sres}} x_{23}(x_{12} + 1) = r_5 \\ r_2, h_9 &\vdash_{\mathsf{sres}} (x_{21} + 1)(x_{23} + 1) = r_6 \end{aligned}$$

Then, using 2-resolution, we get:

$$\begin{aligned} r_1, r_4 &\vdash_{\mathsf{sres}} x_{12} + x_{21} = \ell_1 \\ r_3, r_5 &\vdash_{\mathsf{sres}} x_{12} + x_{23} = \ell_2 \\ r_6, h_5 &\vdash_{\mathsf{sres}} x_{21} + x_{23} + 1 = \ell_3 \end{aligned}$$

Finally, Gaußian elimination discovers $\ell_1 + \ell_2 + \ell_3 = 1$. The maximal degree (or the maximal size) reached in the proof is 2 (resp. 2). These values do not exceed the degree and the size of the input. △

A way how to generate a benchmark of linearly split polynomials whose instances are unsatisfiable is described in Algorithm 5.1. It generates a set of linearly split polynomials A such that $\mathcal{Z}(A) = \emptyset$, and such that there exists a short proof of inconsistency of A using s-resolution. In other words, by applying the s-resolution rule in a specific order on the input and on intermediate results, we obtain 1 after relatively few iterations. Note that s-resolution operates on general linearly split polynomials in this case, and not only on outputs of the standard CNF to ANF conversion.

Algorithm 5.1 `Gen-sres` (Random Decision Trees using sres)

Input: $k, n \in \mathbb{N}_+$.
Output: A set A with $\#A = k + 1$ whose elements are in $\mathbb{L}_n$ and $\mathcal{Z}(A) = \emptyset$.

1: $A = \big\{\{1\}\big\}$
2: **for** $i = 1, \ldots, k$ **do**
3: Select randomly an element R in A and remove R from A.
4: Construct two sets $B_1, B_1 \subset \mathbb{L}_n$ such that $B_1, B_2 \vdash_{\mathsf{sres}} R$.
5: $A := A \cup \{B_1\} \cup \{B_2\}$
6: **end for**
7: **return** A

Proposition 5.37. *Algorithm 5.1 is correct, i.e. it holds $\mathcal{Z}(A) = \emptyset$ for the returned set A for any $k, n \in \mathbb{N}_+$.*

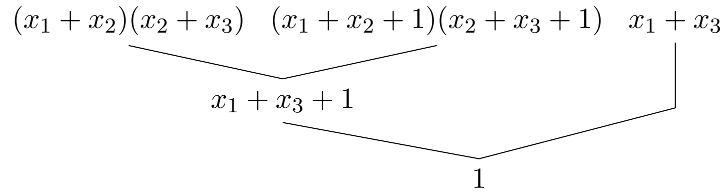
Proof. Let us prove that A is unsatisfiable after each iteration of the for loop. Clearly, we have $A \models \{1\}$ in Step 1. In Step 4, we have $A \cup \{B_1\} \cup \{B_2\} \models A \cup \{R\}$ by the correctness of sres, i.e. the state of A before executing Step 3 is obtained. By iterating the above argument, we get $A \models \{1\}$. □

Note that $A \vdash_{\mathsf{sres}} \{1\}$ by the construction of Algorithm 5.1. Let us illustrate Algorithm 5.1 in some examples and explain how to deal with Steps 3 and 4.

Example 5.38. Let $n = 3$. We start with $A = \big\{\{1\}\big\}$. We remove $\{1\}$ from A and append $B_1 = \{x_1 + x_2\}$ and $B_2 = \{x_1 + x_2 + 1\}$. Note that we have $B_1, B_2 \vdash_{\mathsf{sres}} \{1\}$ by 1-resolution. After that, we obtain $A = \big\{B_1, B_2\big\}$. We remove B_2 and append $B_3 = \{x_1 + x_2 + x_3 + 1,\, x_3 + 1\}$ and $B_4 = \{x_1 + x_2 + x_3,\, x_3\}$. Note that we have $B_3, B_4 \vdash_{\mathsf{sres}} B_2$ by 2-resolution. The resulting set $A = \big\{B_1, B_3, B_4\big\}$ is unsatisfiable by the construction. △

The next example shows, among other things, how to convert arbitrary linear clauses to CNF.

Example 5.39. Consider the linearly split polynomials $f_1 = (x_1 + x_2)(x_2 + x_3)$, $f_2 = (x_1 + x_2 + 1)(x_2 + x_3 + 1)$, and $f_3 = x_1 + x_3$ in $\mathbb{S}_3$. Using the following s-resolution tree, we obtain 1, i.e., $\mathcal{Z}(A) = \emptyset$ for $A = \{f_1, f_2, f_3\}$.



As in Example 4.5, let the sets of clauses C_1, C_2, C_3 encode linear equations $y_1 = x_1 + x_2$, $y_2 = x_2 + x_3$ and $y_3 = x_1 + x_3$ over $\mathbb{F}_2$. The final encoding of A in CNF is then equal to

$$C_1 \cup C_2 \cup C_3 \cup \big\{\{Y_1, Y_2\}, \{\bar{Y}_1, \bar{Y}_2\}, \{Y_3\}\big\}.$$

△

Let us recall that linear clauses generalize clauses, i.e. solvers for sets of linear clauses can admit sets of clauses, too. However, by doing that, we loose the strength of richer representations. E.g., note that almost all transformations (except for S-boxes) used in block ciphers based on substitution-permutation networks are affine (cf. Example 2.36). This means that if we find a representation of the non-linear part of the cipher as a set of linear clauses, we are able to represent the whole the cipher as a set of linear clauses.

Before going to the next example which contains a representation of the S-box of LED using linearly split polynomials, we introduce a DIMACS-like format for sets of linearly split polynomials which allows us to comfortably write down the linear clauses. The first line starts with a header of the form

p lcnf n m,

where n denotes the number of variables, and m denotes the number linearly split polynomials. The next lines contain linearly split polynomials (one per line). Linear polynomials are divided by "|", and zeros in the sequences are read as "+1". Each line ends with "||". For instance, the line

0 1 | 0 2 | 3 4 5 ||

corresponds to the equation

$$(x_1 + 1)(x_2 + 1)(x_3 + x_4 + x_5) = 0.$$

Example 5.40. The S-Box of LED can be represented in the above format as follows.

```
p lcnf 8 29
0 1 | 0 2 | 0 3 4 5 ||
1 | 0 2 | 3 4 6 ||
1 2 | 3 | 0 4 7 ||
1 | 0 2 3 | 0 4 8 ||
0 1 | 2 | 0 3 4 8 ||
0 1 2 | 0 3 | 4 8 ||
0 1 | 2 | 0 3 4 | 0 5 ||
1 2 | 3 | 0 4 | 0 5 ||
1 | 0 2 | 0 3 | 4 5 ||
```

```
0 1 2 | 0 3 | 0 4 | 0 6 ||
0 1 | 0 2 | 3 4 | 0 6 ||
0 1 | 2 | 3 | 4 6 ||
0 1 | 2 3 | 4 | 6 ||
0 1 2 | 0 3 | 0 4 | 7 ||
1 | 0 2 3 | 4 | 0 7 ||
1 | 0 2 | 3 | 0 4 8 ||
0 1 | 0 2 | 3 | 4 8 ||
1 | 0 2 | 3 | 4 | 0 5 ||
0 1 | 2 | 0 3 | 4 | 5 ||
0 1 | 2 | 0 3 | 0 4 | 6 ||
1 | 2 | 0 3 | 4 | 0 6 ||
1 | 0 2 | 0 3 | 0 4 | 0 7 ||
0 1 | 2 | 0 3 | 0 4 | 7 ||
1 | 2 | 0 3 | 4 | 0 7 ||
1 | 2 | 3 | 0 4 | 7 ||
1 | 2 | 5 ||
1 | 2 | 3 | 6 ||
0 1 | 0 2 | 3 | 0 7 ||
0 1 | 0 3 | 4 | 7 ||
```

△

In the above example, we took a CNF containing 64 clauses representing the S-box of LED and searched for patterns which correspond to linearly split polynomials. If a subset of clauses corresponds to a linearly split polynom, it is removed. This process goes on until no suitable subset of clauses is found.

5.5 SRES Closure Algorithms

Now we turn our attention to the fundamental question how to produce SRES-proofs in an algorithmic way. As before, we assume that all sets occurring in the algorithm do not contain duplicates, i.e., that removal of duplicates is applied whenever possible. This is the classical assumption on sets in programming languages such as `python`. We focus on refutations, since many search problems can be reformulated using refutation as in the next example.

Example 5.41. Let S be a set of linearly split polynomials which has a unique solution. Exactly one of the two systems $S \cup \{x_i\}$ and $S \cup \{x_i+1\}$ will be unsolvable, i.e. there exists an SRES-refutation for it. Iterating this procedure for all indeterminates $x_1, \ldots, x_n$ in the system we get the complete solution for S. Note that the total number of refutations is $2n$. △

When computing s-resolvents, it is necessary to use the maximal possible s, since otherwise $\ell_i(\ell_i + 1) = 0$ forces the resolvent to be zero (see Example 5.20). In order to make everything precise, we spell out an explicit algorithm for computing s-resolvents, namely Algorithm 5.3 and its subroutine Algorithm 5.2.

Algorithm 5.2 `AnalyzePair` (Analysis of a Pair)

Input: Sets $F, G \subseteq \mathbb{L}_n$.
Output: Sets $A_F, B, F', G' \subseteq \mathbb{L}_n$.

1: $A_F := \{\ell \in F \mid \ell + 1 \in G\}$
2: $A_G := \{\ell \in G \mid \ell + 1 \in F\}$
3: $B := \{\ell \in F \mid \ell \in G\}$
4: $F' := F \setminus (A_F \cup B)$
5: $G' := G \setminus (A_G \cup B)$
6: **return** (A_F, B, F', G')

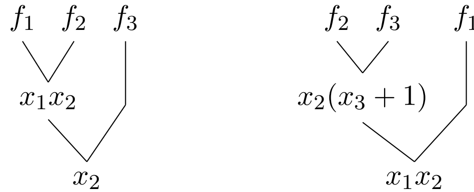
Algorithm 5.2 decompose sets F, G into the set A_F of the "complementary" linear polynomials, into the set B of the "shared" linear polynomials, and the "remainders" F', G'.

Example 5.42. Let $F = \{x_1, x_2, x_3, x_4\}$ and $G = \{x_1 + 1, x_2 + 1, x_3, x_5 + 1\}$. Algorithm 5.2 decomposes F and G into $A_F = \{x_1, x_2\}$, $B = \{x_3\}$, $F' = \{x_4\}$ and $G' = \{x_5 + 1\}$. △

The rule sres is implemented in Algorithm 5.3. This algorithm is clearly finite, and the proof of correctness follows directly from Definition 5.16.

Given three or more linearly split polynomials, the order of performing s-resolution steps does matter, as the following example shows.

Example 5.43. Let $f_1 = x_1x_2x_3$, $f_2 = x_1x_2(x_3 + 1)$, and $f_3 = x_1 + 1$. Then we may draw the following 1-resolution trees.



Clearly, the polynomial x_2 is more useful than x_1x_2. Thus the order of the s-resolution steps may influence the quality of the output. △

Any set of clauses corresponds to a set of linearly split polynomials via Algorithm 4.2 where the linear polynomials are of the form x_i and $x_i + 1$. That is why we can look into refuting a set of clauses using SRES as a separate case. The idea of the algorithms in this section is to compute a closure using sres. The definition of the sres-closure is given below (see Definition 2.24).

Definition 5.44. Let $C \subset \mathbb{L}_n$. The sres-closure of C is defined recursively as follows.

(a) $\mathrm{SRes}^0(C) = C$

Algorithm 5.3 `Sres` (s-Resolution of Two Polynomials)

Input: Sets $F, G \subseteq \mathbb{L}_n$.

Output: A number $s = \#\{\ell \in F \mid \ell + 1 \in G\} \geq 1$ and $R \subseteq \mathbb{L}_n$ such that R is the s-resolvent of F and G. If an s-resolvent of F and G cannot be constructed for any s, the algorithm outputs $s = 0$ and $R = \emptyset$.

Require: Algorithm 5.2.

1: $(A_F, B, F', G') :=$ `Analyze`(F, G)
2: **if** $A_F = \emptyset$ **then**
3:   **return** $(0, \emptyset)$
4: **else**
5:   Write A_F as $\{\ell_1, \dots, \ell_s\}$.
6:   $R := B \cup F' \cup G'$
7:   **if** $s = 1$ **and** $R = \emptyset$ **then**
8:     **return** $(1, \{1\})$
9:   **else**
10:     **for** $i = 1, \dots, s-1$ **do**
11:       **if** $\ell_i + \ell_{i+1} \notin R$ **then**
12:         $R := R \cup \{\ell_i + \ell_{i+1} + 1\}$
13:       **else**
14:         **return** $(s, \{0\})$
15:       **end if**
16:     **end for**
17:   **end if**
18:   **return** (s, R)
19: **end if**

(b) $\mathrm{SRes}^1(C) = C \cup \{r \mid r \text{ is an } s\text{-resolvent of two clauses in } C \text{ for some } s \geq 1\}$

(c) $\mathrm{SRes}^{i+1}(C) = \mathrm{SRes}(\mathrm{SRes}^i(C))$ for $i > 1$

(d) The set $\mathrm{SRes}^\infty(C) = \bigcup_{i=0}^{\infty} \mathrm{SRes}(C)$ is called the **sres-closure** of C.

In the next proposition, `StdANF` denotes Algorithm 4.1 which converts a set of clauses to linearly split polynomials. Proposition 5.45 is the key for proving completeness of Algorithm 5.4.

Proposition 5.45. *Let C be a set of clauses.*

(a) We have $\mathrm{SRes}^\infty(\mathtt{StdANF}(C)) \supseteq \mathtt{StdANF}(\mathrm{Res}^\infty(C))$.

(b) Let $S = \{F_1, \dots, F_m\}$ such that $F_i \subset \mathbb{L}_n$ and every linear polynomial $\ell \in F_i$ satisfies $\#\operatorname{Var}(\ell) = 1$ for $i = \{1, \dots, m\}$. Then we have $S \models \{1\}$ if only if $\{1\} \in \mathrm{SRes}^\infty(S)$.

Proof. Claim (a) follows from Propositions 4.6 and 5.19. Claim (b) follows from Proposition 2.25 and Claim (a). □

Algorithm 5.4 is inspired by the design of Algorithm 2.1. The pairs (F,G) in Algorithm 5.4 are processed in the increasing order with respect to the following ordering relation.

Definition 5.46. Let $F, G, F_1, F_2, G_1, G_2 \subseteq \mathbb{L}_n$.

(a) We write $F \preceq G$ if we have $\deg(F) < \deg(G)$, or if we have $\deg(F) = \deg(G)$ and $\text{size}(F) < \text{size}(G)$.

(b) Let $s_1 = \#\{\ell \in F_1 \mid \ell + 1 \in G_1\} \geq 1$ and $s_2 = \#\{\ell \in F_2 \mid \ell + 1 \in G_2\} \geq 1$. We write $(F_1, G_1) \trianglelefteq (F_2, G_2)$ if $R_1 \preceq R_2$ where R_1 is the s_1-resolvent of F_1 and G_1, and R_2 is the s_2-resolvent of F_2 and G_2.

Note that the s-resolvents in (b) exist because of the inequalities $s_1 \geq 1$ and $s_2 \geq 1$. The SRES closure algorithm for a set of clauses is given in Algorithm 5.4.

Algorithm 5.4 `SRES-Refute-CNF` (The SRES Refutation Algorithm for CNF)

Input: Subsets $F_1, \ldots, F_m \subseteq \mathbb{L}_n$ such that any linear polynomial ℓ in F_i satisfies $\#\operatorname{Var}(\ell) = 1$ for $i \in \{1, \ldots, m\}$. Furthermore, the set F_i is the set of linear factors of a linearly split polynomial for $i \in \{1, \ldots, m\}$.
Output: `False` if $F_1, \ldots, F_m \models \{1\}$, `True` otherwise.
Require: Algorithm 5.3.

1: $S := \{F_1, \ldots, F_m\}$
2: **if** $\{1\} \in S$ **then**
3:   **return** `False`
4: **end if**
5: Let P be the list of all (F_i, F_j) such that $1 \leq i < j \leq m$.
6: **while** $P \neq \emptyset$ **do**
7:   Let (F,G) be a minimum of P w.r.t. $\trianglelefteq$, and remove (F,G) from P.
8:   (s, R) :=`Sres`(F,G)
9:   **if** $s \neq 0$ **then**
10:     **if** $R = \{1\}$ **then**
11:       **return** `False`
12:     **else if** R is not a subset of any $Q \in S$ **then**
13:       Append all (R,G) such that $G \in S, R \neq G$ at the end of the list P.
14:       $S := S \cup \{R\}$
15:     **end if**
16:   **end if**
17: **end while**
18: **return** `True`

Proposition 5.47. *Algorithm 5.4 is finite and correct.*

Proof. Let us prove finiteness first. Note that the set R produced in Step 8 is the set of linear factors of a linearly split polynomial. Since there exist only finitely many linearly

split polynomials of degree $\leq n$, the set S can be enlarged only finitely many times. The number of all pairs formed from sets in S is finite as well. The while-loop is finite, because there are only finitely many pairs in P. Note that P does not contain duplicities because of Step 12. Thus is follows that the entire algorithm is finite.

Correctness follows from Proposition 5.45 and from the fact that all possible s-resolvents are created in Step 8, and hence all elements in $\mathrm{SRes}^{\infty}(\{F_1, \ldots, F_m\})$ are successively derived. □

It is clear that s-resolution is more powerful for sets of clauses which share a lot of variables. Therefore Algorithm 4.3 can be used profitably as a preprocessing step for Algorithm 5.4. This combination can be seen as a lightweight version of Algorithm 4.4 because s-resolvents are build instead of S-polynomials. Let us see a case in point.

Example 5.48. Consider the linearly split polynomials $h_1 = x_1x_2x_3$, $h_2 = (x_1 + 1)(x_2 + 1)x_3$, and $h_3 = x_3 + 1$. Then the set $\{h_1, h_2, h_3\}$ corresponds to a 1-block (see Sect. 4.4), and s-resolution discovers the linear polynomial $\ell = x_1 + x_2 + 1$. Note that the degree (or the size) of ℓ is 1 (or 2). △

In the next example we compare Algorithm 5.4 to proofs obtained by the proof system ERES in Definition 5.7.

Example 5.49. Let $C = \{c_1, c_2\}$ where $c_1 = \{L_1, L_2\}$ and $c_2 = \{\bar{L}_1, \bar{L}_2\}$ are clauses with literals $L_1 = X_1$ and $L_2 = X_2$. Applying 2-resolution, we get the clause $c_3 = \{L_3\}$ with a new variable L_3 with the meaning $L_3 \Leftrightarrow X_1 \oplus X_2$. Note that, unlike in ERES, we do not explicitely encode the definition $L_3 := X_1 \oplus X_2$ in CNF. △

Let us focus our attention to general linearly split polynomials, and not only a set of clauses. First of all, we point out that the rule weak in Definition 5.21 is very important in the general case. For Algorithm 5.4 to be correct, it is essential that every input polynomial of Algorithm 5.4 is the standard algebraic representation of a clause. The following example shows that sres (only by itself) cannot detect unsatisfiability of general sets of linearly split polynomials.

Example 5.50. Let us apply Algorithm 5.4 to the linearly split polynomials $h_1 = (x_1+x_2)(x_3+x_4+1)$, $h_2 = x_1$, $h_3 = x_2+1$, $h_4 = x_3$ and $h_5 = x_4$ in $\mathbb{S}_4$. Since we cannot form any s-resolvents, the algorithm returns `True`. However, the equality

$$1 \;=\; h_1 + h_2 + h_3 + (x_1 + x_2)h_4 + (x_1 + x_2)h_5$$

shows that the system is inconsistent, i.e., that $\mathcal{Z}(\langle h_1, \ldots, h_5 \rangle) = \emptyset$. △

The previous example illustrates that the weakening rule cannot be omitted. Algorithm 5.5 extends two polynomials by weak in all possible ways such that s-resolution can be applied. More precisely, this set of extensions is defined as follows.

Definition 5.51. Let $F, G \subseteq \mathbb{L}_n$. The set of all pairs $K \subseteq \mathbb{L}_n \times \mathbb{L}_n$ such that the following two conditions hold for all $(F', G') \in K$.

(a) $\#\{\ell \in F' \mid \ell + 1 \in G'\} \geq 1$

(b) $F' \cup G' \subseteq F \cup G \cup \{\ell + 1 \mid \ell \in F \cup G\}$

is called the **set of expansions** for F, G.

Condition (a) encodes the fact that there exists at least one $\ell \in \mathbb{L}_n$ in F' and G' on which we can s-resolve. Condition (b) restrains F', G' to contain linear polynomials ℓ or $\ell + 1$ such that $\ell \in F \cup G$. Note that the set of expansions for F, G is unique.

Algorithm 5.5 produces the set of expansions in a direct way, i.e., Condition (a) is implemented in Step 10, and Condition (b) is fulfilled because of the two foreach loops in Steps 3, 4.

Algorithm 5.5 `AllExpansions` (All Possible Expansions to s-Resolvents)

Input: Sets $F, G \subseteq \mathbb{L}_n$.
Output: The set of expansions for F, G.

1: $\{\ell_1, \ldots, \ell_k\} := \{\ell \in F \mid \ell \notin G, \ell + 1 \notin G\}$
2: $\{\ell'_1, \ldots, \ell'_{k'}\} := \{\ell \in G \mid \ell \notin F, \ell + 1 \notin F\}$
3: **foreach** $A \in \{\ell_1, \ell_1 + 1, 1\} \times \cdots \times \{\ell_k, \ell_k + 1, 1\}$ **do**
4:   **foreach** $B \in \{\ell'_1, \ell'_1 + 1, 1\} \times \cdots \times \{\ell'_{k'}, \ell'_{k'} + 1, 1\}$ **do**
5:     Write $A = (a_1, \ldots, a_k)$.
6:     Write $B = (b_1, \ldots, b_{k'})$.
7:     $F' := F \cup \bigcup_{i=1}^{k} \{b_i\}$
8:     $G' := G \cup \bigcup_{i=1}^{k'} \{a_i\}$
9:     Minimize the representation of F' and G' by applying unit cancellation, i.e., remove the element 1 from F', G'.
10:     **if** $\#\{\ell \in F' \mid \ell + 1 \in G'\} \geq 1$ **then**
11:       $K := K \cup \{(F', G')\}$
12:     **end if**
13:   **end foreach**
14: **end foreach**
15: **return** K

The next example shows the generation of the pairs in Algorithm 5.5.

Example 5.52. Let $F = \{x_1, x_2, x_3 + 1\}$ and $G = \{x_1 + 1, x_2, x_4\}$. Then we may extend F to F itself, to $F_1 = \{x_1, x_2, x_3 + 1, x_4\}$, or to $F_2 = \{x_1, x_2, x_3 + 1, x_4 + 1\}$. Similarly, we may extend G to G, to $G_1 = \{x_1 + 1, x_2, x_4, x_3 + 1\}$, or to $G_2 = \{x_1 + 1, x_2, x_4, x_3\}$. Altogether, nine pairs $(F, G), (F, G_1), (F, G_2), (F_1, G), (F_1, G_1), (F_1, G_2), (F_2, G), (F_2, G_1), (F_2, G_2)$ are constructed. △

Now we are ready to present Algorithm 5.4 for constructing SRES-refutations.

Proposition 5.53. *Algorithm 5.6 is correct, refutationally complete, and finite. In particular, the algorithm returns* `False` *if and only if* $F_1, \ldots, F_m \models \{1\}$.

Algorithm 5.6 `SRES-Refute` (The SRES Refutation Algorithm)

Input: Subsets $F_1, \ldots, F_m \subseteq \mathbb{L}_n$ such that F_i is the set of linear factors of a linearly split polynomial.
Output: `False` if $F_1, \ldots, F_m \models \{1\}$, `True` otherwise.
Require: Algorithms 5.3, 5.5.

1: $S := \{F_1, \ldots, F_m\}$
2: **if** $\{1\} \in S$ **then**
3:    **return** `False`
4: **end if**
5: Apply unit cancellation to S.
6: $\{Q_1, \ldots, Q_k\} := S \cup \bigcup_{i=1}^{n} \{x_i, x_i + 1\}$
7: Let P be the list containing all pairs computed by `AllExpansions`(Q_i, Q_j) for $1 \leq i < j \leq k$.
8: **while** $P \neq \emptyset$ **do**
9:    Let (F, G) be a minimum of P w.r.t. $\trianglelefteq$, and remove (F, G) from P.
10:    $R :=$`Sres`(F, G)
11:    **if** $R = \{1\}$ **then**
12:       **return** `False`
13:    **else if** R is not a subset of any $Q \in S$ **then**
14:       Remove all Q from S with $R \subsetneq Q$ and all pairs (Q_1, Q_2) from P such that $R \subsetneq Q_1$ or $R \subsetneq Q_2$.
15:       Append all pairs in `AllExpansions`(R, G) for $G \in S, R \neq G$ to the list P.
16:       $S := S \cup \{R\}$
17:    **end if**
18: **end while**
19: **return** `True`

Proof. Finiteness follows from the fact that there are only finitely many linear polynomials in $\mathbb{L}_n$. Hence the sets in S are finite, and so is the set $P \subseteq \mathbb{L}_n \times \mathbb{L}_n$.

The algorithm is correct because the SRES inference rules are sound by Corollary 5.19.

It remains to show that the algorithm is refutationally complete. Assume that the ideal generated by the polynomials corresponding to the input sets has no common zero. By Proposition 5.30, we know that there exists an SRES-refutation of $F_1, \ldots, F_m$. We show that this SRES-refutation can be found by the algorithm. We prove it by a more general claim, namely, that Algorithm 5.6 constructs all possible proofs starting from the initial premises.

The Boolean axiom is incorporated in Step 6, and consequences of the weakening rule are constructed in the function `AllExpansions` such that all possible choices to form an s-resolvent are created for all possible $s \in \mathbb{N}_+$. The s-resolution rule is then applied by calling `Sres`. All possible SRES-derivations are produced by the mechanism of appending the new pairs in Step 15.

Note that any set Q that is a proper superset of some set already occurring in S can be skipped because all possible s-resolvents that would be created using Q are at that

time already in P. Thus the algorithm sequentially creates all SRES-derivations from $F_1, \ldots, F_m$. □

Algorithm 5.6 arranges the computation such that "smaller" sets in terms of size are preferred. The list P in Algorithm 5.6 can be implemented as a min-heap such that the minimal pair is always easily found and extracted. The number of pairs in P may be huge. Thus it is convenient to compute the s-resolvents only in Step 10. The next example indicates how the pairs can be stored. The s-resolvent is created only if its size is minimal.

Example 5.54. Let $F_1 = \{x_1 + 1\}$ and $F_2 = \{x_1, x_2\}$. Algorithm `AllExpansions` outputs (F_1, F_2), $(F_1 \cup \{x_2 + 1\}, F_2)$, $(F_1 \cup \{x_2\}, F_2)$. The pair $(F_1 \cup \{x_2\}, F_2)$ can be stored as a tuple $(1, \{x_2\}, 2, \emptyset, 1)$ with the meaning "the s-resolvent of $F_1 \cup \{x_2\}$ and $F_2 \cup \emptyset$ has size 1". △

Clearly, we need to know the sizes of all s-resultants in P in order to select the minimum. Thus the chosen format is very convenient because the size of the s-resolvent can be predicted as in [54, Alg. 8] without computing the actual s-resolvents.

Furthermore, one can form only 2-resolvents in Algorithm 5.6 since 2-resolution is enough to simulate RLIN which is implicationally and refutationally complete. However, "extra" linear clauses coming from s-resolution steps with $s \geq 3$ may come in handy, and the refutation can be found faster in Algorithm 5.6.

5.6 SRES Refutation Using DPLL Techniques

It took several decades to develop efficient SAT solvers based on classical resolution. Algorithm 5.6 uses more advanced reasoning than classical resolution. However, it is guided by a very simple strategy, namely "derive narrow polynomials first". Algorithm 5.6 is not very practical because a huge number of pairs is created in `AllExpansions`. One way how of supplanting weakening, which is very important for computing the sres-closure, is to turn the algorithm into a DPLL-based version.

Boolean constraint propagation (see Algorithm 2.4) is the key ingredient of modern SAT solvers. Thus we define a new propagation mechanism, called *Gaußian constraint propagation* (GCP), which is geared towards our setting. GCP is described in Algorithm 5.7. The set D contains assignments of indeterminates and linear polynomials, i.e. it can be represented by a decision stack (see Definition 2.27 and Example 2.28). In this setting, a decision stack is a two-dimensional tuple of linear polynomials, e.g. the linear polynomial $x_1 + 1$ denotes the decision $x_1 \mapsto 1$, and the linear polynomial $x_1 + x_2$ denotes the equation $x_1 + x_2 = 0$. New linear polynomials which are returned by Algorithm 5.7 are referred to as *implied* linear polynomials. Notice that in Algorithm 5.7 the linear polynomial ℓ rewrites all indeterminates in ℓ' and not only $\mathrm{LT}(\ell')$.

Proposition 5.55. *Algorithm 5.7 is correct, i.e. if the algorithm returns $a =$ `False` then we have $\mathcal{Z}(S|_D) = \emptyset$. Furthermore, for its output D' we have $S \vdash_{\mathsf{SRES}} \{H\}$ for all $H \in D'$.*

Algorithm 5.7 `GCP` (Gaußian Constraint Propagation)

Input: A set of linearly LT_σ-interreduced linear polynomials D, a set $S = \{F_1, \ldots, F_m\}$ with $F_i \subseteq \mathbb{L}_n$, a term ordering σ.

Output: A set of LT_σ-interreduced linear polynomials D' such that $S|_D \models S|_{D'}$, a Boolean value a. If $a = \texttt{False}$, then we have $\mathcal{Z}(S|_D) = \emptyset$.

1: $D' := \emptyset$
2: **repeat**
3: $E := D'$
4: **for** $F \in S$ **do**
5: $F' := F$
6: **while** $\ell \in D \cup D'$ such that $\mathrm{LT}_\sigma(\ell) \in \mathrm{Supp}(\ell')$ for some $\ell' \in F'$ **do**
7: Rewrite the linear polynomial ℓ' in F' using the rewriting rule $\mathrm{LT}_\sigma(\ell) \mapsto \ell - \mathrm{LT}_\sigma(\ell)$.
8: **end while**
9: $r := \prod_{\ell \in F'} \ell$
10: **if** $r \in \mathbb{L}_n$ **and** $r \neq 0$ **then**
11: **if** $r = 1$ **then**
12: **return** $(D', \texttt{False})$
13: **else**
14: $D' := D' \cup \{r\}$
15: **end if**
16: **end if**
17: **end for**
18: **until** $E = D'$
19: **return** $(D', \texttt{True})$

Proof. The claims follows from the fact that the substitution of linear polynomials is sound and from Proposition 5.28 and 5.25. □

Notice that to design a combination of the closure algorithm given in Algorithm 5.6 with Algorithm 5.7 such that the resulting algorithm is refutationally complete is not an easy task. An instance that demonstrates some issues which have to be taken care of is given in the next example.

Example 5.56. Consider the set of linearly split polynomials $S = \{h_1, \ldots, h_5\}$ with $h_1 = x_1(x_1 + x_2)$, $h_2 = x_2(x_1 + x_2)$, $h_3 = x_1x_2$, $h_4 = (x_1 + x_2 + x_3 + 1)(x_3 + 1)$ and $h_5 = (x_2 + x_3)(x_1 + x_3)$. Note that the rule sres cannot be applied, and no linear polynomial can be substituted using Algorithm 5.7. However, the system $\{h_1, \ldots, h_5\}$ is inconsistent. This can be proven as follows. From h_1, h_2, h_3 we get that $x_1 = x_2 = 0$. Then we obtain $x_3 = 1$ by the substitution of $x_1 = x_2 = 0$ into h_4. However, the partial assignment is not consistent with h_5, i.e. the polynomial 1 cannot be derived by an application of sres and Algorithm 5.7. △

However, using Algorithm 5.7 in the DPLL structure given in Algorithm 2.6 gives us

the refutationally complete Algorithm 5.8. Note that the structure of Algorithm 2.6 is very similar to Algorithm 5.8.

Algorithm 5.8 `SRES-DPLL` (The SRES Refutation based on DPLL)

Input: A set $S = \{F_1, \dots, F_m\}$ with $F_i \subseteq \mathbb{L}_n$, a term ordering σ.
Output: A LT_σ-interreduced tuple of linear polynomials $D \subseteq \mathbb{L}_n$ such that $\mathcal{Z}(D) \subseteq \mathcal{Z}(S)$ if $\#\mathcal{Z}(S) > 0$; `False` otherwise.
Require: Algorithms 5.7 and 2.5.

1: $(D', a) = \texttt{GCP}(\emptyset, S, \sigma)$
2: **if** $a =$ `False` **then**
3:   **return** `False`
4: **end if**
5: $D := (D')$
6: $A := \emptyset$; $b :=$ `True`
7: **while** D does not contain n LT_σ-interreduced polynomials **do**
8:   **if** $b =$ `True` **then**
9:     Choose an indeterminate $x_i \in \mathrm{Var}(S) \setminus \left\{ \mathrm{LT}_\sigma(\ell) \mid \ell \in \bigcup_{T \in D} T \right\}$ and a Boolean value $v \in \{0, 1\}$.
10:     Append the tuple $(x_i + v)$ to D
11:     $A := A \cup \{x_i + v\}$
12:   **end if**
13:   $b :=$ `True`
14:   $(a, D') = \texttt{GCP}(D, S, \sigma)$
15:   Append the linear polynomials in D' to the end of the last tuple in D.
16:   **if** $a =$ `False` **then**
17:     $(D, d) := \texttt{Chro-Backtrack}(A, D)$.
18:     $b :=$ `False`
19:     **if** $d =$ `False` **then**
20:       **return** `False`
21:     **end if**
22:   **end if**
23: **end while**
24: **return** D

The correctness of the algorithm follows the same lines as the proof for the classical DPLL procedure. (For a formal proof of DPLL, see [82, Th. 1].) The only differences are the different propagation mechanism (`GCP` instead of `BCP`) and the different data structure (linear clauses instead of clauses). The branching step is based on the iterative decomposition of S into the two sets $x_i + 1$, $S|_{x_i \mapsto 1}$ and x_i, $S|_{x_i \mapsto 0}$. Every zero $a \in \mathcal{Z}(S)$ can be found in the union of the solution sets of the above systems.

Proposition 5.57. *Algorithm 5.8 is correct, refutationally complete, and finite. In particular, the algorithm returns* `False` *if and only if* $F_1, \dots, F_m \models \{1\}$.

Proof. Let us prove finiteness. The algorithm chooses at most n indeterminates in

Step 9. The procedure `Chro-Backtrack` enumerates all possible assignments for those indeterminates, i.e. after at most 2^n iterations the while-loop terminates.

The list D contains LT_σ-interreduced linear polynomials in every iteration of the algorithm. In particular, n linear polynomials in D defines a zero $a \in \mathcal{Z}(D)$. By Proposition 5.55 we have $a \in \mathcal{Z}(S)$. Conversely, $\mathcal{Z}(D) = \emptyset$ implies $\mathcal{Z}(S) = \emptyset$, i.e. correctness follows from Proposition 5.55.

It remains to show that the algorithm is refutationally complete. Assume that $\mathcal{Z}(S) = \emptyset$. This means that the set $S|_{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n}$ contains the polynomial 1 for every value $v_1, \ldots, v_n \in \{0, 1\}$. The subroutine `GCP` eliminates the partial assignments of the indeterminates $x_1, \ldots, x_n$ which cannot be extended to a common zero. That is why, after backtracking to decision level 0, the algorithm returns `False`. □

As a next step, the above DPLL-based algorithm could be modified to a CDCL-like procedure. In the next example, we give some suggestions inspired by 1UIP learning for our case. Please refer to Example 2.30 for a comparison and the notation.

Example 5.58. Let $c_1 = \{x_1+x_2+x_3+1,\ x_3+x_4+x_5\}$, $c_2 = \{x_2+x_4+x_5,\ x_5+x_6\}$, and $c_3 = \{x_5+x_6+1,\ x_7\}$. Assume that the decisions are as follows (in order of appearance) $x_1 \mapsto 1$, $x_2 \mapsto 0$, $x_7 \mapsto 1$, $x_3 \mapsto 1$. After each decision is made, we run GCP to derive implied linear polynomials. The corresponding decision stack is given below.

decision level	*decision*	*implied linear polynomials*
0		
1	$x_1 + 1$	
2	x_2	
3	$x_7 + 1$	$x_5 + x_6 + 1\,(c_3)$, $x_4 + x_6 + 1\,(c_2)$
4	$x_3 + 1$	$1\,(c_1)$

Firstly, note that it is not possible to s-resolve c_1 and c_2. The learned linearly split polynomial is the 1-resolvent of c_2 and c_3, i.e. $r = \{x_2 + x_4 + x_5 \,@\, 2,\ x_7 \,@\, 3\}$. Note that we suggest labeling the new linear polynomials in r by the decision levels of the linear polynomials in the decision stack which has the same leading terms as the linear polynomials in r. Secondly, r is not a 1UIP in the common sense because there is no linear polynomial in r which was assigned at level 4. The set r can be appended to the input linearly split polynomials because we have $\mathcal{Z}(c_1, c_2, c_3) = \mathcal{Z}(c_1, c_2, c_3, r)$ by Corollary 5.19. △

Translation of non-chronological backtracking from sets of clauses to sets of linear clauses is not straightforward. However, it is possible to combine the learning procedure described in Example 5.58 with chronological backtracking given by Algorithm 2.5. This combination is sound and complete due to Proposition 5.57 and Corollary 5.19.

5.7 Experiments

In this section we examine the efficiency of the several s-resolution techniques to the classical resolution. All tests were executed on a computing 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 chapter were prototypically implemented in `python` version 2.7. For testing SAT solvers, we use `CryptoMiniSat` [106] version 5.6.6 with enabled Gaußian elimination, `Glucose` [6] version 4.1 and `MiniSat` [41] version 2.2.0. All SAT solvers are run out-of-the-box without any special parameter tuning. The timeout for all tests was set to 1200 or 1500 seconds. If an execution exceeds the limit, it is marked in the tables by ">1500" or by "-" if the timeout was set to 1200 seconds.

Refuting CNF Formulas Using s-Resolution

In Table 5.1 we compare Algorithm 5.4 to the resolution closure algorithm. By the *resolution closure algorithm* (RCA), we mean Algorithm 5.4 with the following restriction: Algorithm 5.3 returns a value only if $s = 1$, i.e. s-resolvents with $s \geq 2$ are ignored. Hence both algorithms share the same structural framework. Algorithm 5.4 produces far more s-resolvents than the RCA, because s is not restricted to 1. As a consequence, Algorithm 5.4 creates the same intermediate results as RCA together with some new linearly split polynomials. To derive these extra polynomials first, the choice in which order the pairs are processed in Step 4 is crucial.

The instances `tseitin-`i`-`j correspond to Tseitin formulas derived from a random grid graph of size $i \times j$ such that the resulting formula is unsatisfiable (see [76, Sect. 2]). The instances `aim` correspond to random 3-SAT formulas in [4] and `dubois` to random CNF formulas by Olivier Dubois [1]. The instance `php-`i`-`j corresponds to a pigeonhole-principle formula, namely placing i pigeons into j holes (see [76, Sect. 2]). Table 5.1 contains the timings (in seconds) for RCA and for Algorithm 5.4 on the given instances, the number of linearly independent linear polynomials discovered before 1 is found during the run of Algorithm 5.4, as well as the cpu timing of `CryptoMiniSat` (abbreviated to `CMS`), `Glucose` and `MiniSat` (abbreviated to `MS`).

Notice that Algorithm 5.4 uses Gaußian elimination to show that 1 is in the ideal generated by the given linearly split polynomials. So, as expected, it is not generally faster on pigeonhole principle formulae. Rather, it tends to find refutations faster than the resolution closure algorithm provided the instances have a rich linear structure.

Whereas resolution closure algorithms tend to produce many redundant clauses (see [27, Sect. 4.3.1]), resolvents learned in CDCL are non-redundant, i.e., the learned clauses are not implied by smaller clauses with respect to the ordering of the literals. Algorithm 5.4 uses more advanced reasoning than classical resolution. However, it is guided by a very simple strategy, namely "derive narrow polynomials first". Clearly, Algorithm 5.4 in its current form does not reach the versatility of modern CDCL-based solvers. For instance, `CryptoMiniSat` is able to solve all instances in Table 4.1 and Table 5.1 within a few seconds. However, on the two largest problems in Table 5.1, Algorithm 5.4 does actually outperform `Glucose`. Furthermore, Algorithm 5.4 outperforms `MiniSat` on all largest instances of `tseitin` in Table 5.1. Because `Glucose` and `MiniSat` achieve comparable results, we can see that the instances of `tseitin` are favourable to algebraic processing.

[1] The latter two benchmarks are available at: `www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

Table 5.1: The execution time of RCA, Algorithm 5.4, and SAT solvers

Instance	CNF		RCA	Algorithm 5.4		`CMS`	`Glucose`	`MS`
	#vars	#clauses	sec	sec	#lin	sec	sec	sec
`aim-100-1-6-no-1`	100	160	5.25	3.8	12	0.01	0.0	0.0
`aim-100-2-0-no-1`	100	200	17.84	38.91	5	0.01	0.0	0.0
`php-4-3`	12	22	0.96	35.44	6	0.01	0.0	0.0
`php-5-3`	15	35	3.95	369.43	9	0.01	0.0	0.0
`php-8-3`	24	92	209.59	>1500		0.01	0.0	0.0
`dubois22`	66	176	29.52	0.34	44	0.01	0.0	0.0
`dubois25`	75	200	33.22	0.42	50	0.01	0.0	0.0
`dubois30`	90	240	45.32	0.77	60	0.02	0.0	0.0
`dubois50`	150	400	97.26	1.64	100	0.01	0.0	0.0
`tseitin-2-15`	43	112	194.41	0.15	30	0.01	0.0	0.0
`tseitin-2-30`	88	232	274.07	0.73	60	0.01	0.0	0.0
`tseitin-2-40`	118	312	322.18	1.11	80	0.01	0.0	0.0
`tseitin-8-9`	127	448	>1500	54.68	72	2.12	0.47	83.22
`tseitin-9-9`	144	512	>1500	82.71	81	1.61	1.19	262.00
`tseitin-10-10`	180	648	>1500	184.02	100	1.3	5.95	>1500
`tseitin-10-11`	199	720	>1500	241.09	110	1.33	31.63	>1500
`tseitin-11-11`	220	800	>1500	338.08	121	1.3	92.97	>1500
`tseitin-11-12`	241	880	>1500	450.63	132	1.22	52.16	>1500
`tseitin-12-12`	264	968	>1500	651.68	144	1.27	1033.3	>1500
`tseitin-13-13`	312	1152	>1500	1002.96	169	1.21	1200.01	>1500

s-Resolution vs. CDCL SAT Solving

The purpose of Algorithm 5.4 is to study enhanced reasoning into which DPLL or CDCL techniques can be incorporated. As a next step, to make the current method more powerful, we need to combine modern backtracking-based search algorithms for SAT with s-resolution, or with sophisticated techniques involving Gaußian elimination such as in [105]. Note that Algorithm 5.4 targets instances containing many `XOR` constraints. Because CNF encodings of many symmetric cryptosystems, e.g., of substitution-permutation networks, contain a large number of linear relations, Algorithm 5.4 and its improvements can be used in algebraic cryptanalysis of these ciphers.

Below we present a class of examples where s-resolution with $s \geq 1$ is superior to classical resolution. These examples are generated by Algorithm 5.1. To encode the resulting sets of linearly split polynomials in CNF, we used the same approach as in Example 5.39.

Some results for benchmark examples of the above shape are presented in Table 5.2. It contains the number of decisions (# decisions), both for `CryptoMiniSat` with enabled Gaußian elimination and for `Glucose`, as well as the number of applications of the s-resolution rule (# steps) that is sufficient to derive the contradiction. The symbol "-" denotes that the item is unknown due to exceeding the timeout. Because the number of decisions bounds the length of resolution proofs, the table gives us an overview how difficult the instances for the classical resolution are. However, there exist short proofs for the given instances in SRES. Note that we do not claim anything about the computability of these proofs yet. For instance, the current version of Algorithm 5.4 cannot find these

short s-resolution proofs within the timeout.

Table 5.2: The evaluation of CDCL methods on an s-resolution benchmark

CNF		`CryptoMiniSat`	`Glucose`	s-res
#vars	#clauses	#decisions	#decisions	#steps
308	8905	987619	2164862	100
1262	8429	3266516	5011847	130
1661	11267	4680191	6672070	110
1961	13331	-	-	150
2261	15565	9817729	14163825	170
2277	15615	5440028	8478117	190
2692	18535	1351701	2075519	210
2805	19339	-	10824240	230
3679	25445	5638667	6952647	250
3923	27203	5444029	10151267	270
3628	25065	787123	1308057	290
4061	27999	4012913	6240094	330
4105	28285	3960929	5547085	310
6401	43367	-	-	400

The instances in Table 5.2 are rather hard for resolution-based solvers. However, there is no guarantee that the used SAT solvers found the shortest refutations.

This section finishes with some experiments using the above benchmark where the SRES-proofs are actually computed by Algorithm 5.8. The instances are generated by Algorithm 5.1 with the following parameters: the number of indeterminates n, the maximal length of linear polynomials a, and the parameter k defined in the description of Algorithm 5.1. Table 5.3 contains the timings (in seconds) for `CryptoMiniSat` with enabled Gaußian elimination and `Glucose`. The left-most column provides the timings of Algorithm 5.8.

From the results in Table 5.3 we see that the `python` implementation of Algorithm 5.8 is already in the range of the highly optimised C++ implementations of the SAT solvers. In comparison to `CryptoMiniSAT` and `Glucose`, Algorithm 5.8 does not apply any form of learning. Note that even if the benchmark is produced by an application of sres, Algorithm 5.8 does not have any knowledge about the order of their applications or what “right” decisions it should make. Nevertheless, our algorithm does not perform in a very stable way yet. Even a slight modification of the parameters of the instances may change its behaviour completely. We think this is due to our branching heuristic which is still very preliminary. To become competative with other solvers, Algorithm 5.8 has to keep up with the implementation details which make CDCL solvers powerful, e.g. efficient lazy representations for linearly split polynomials, learning schemes (such as one described in Example 5.58), and an efficient implementation of Algorithm 5.7 have to be applied.

Table 5.3: The evaluation of Algorithm 5.8 and SAT solvers on the benchmark generated by Algorithm 5.1

Instances			CNF		Alg. 5.8	`CMS`	`Glucose`
n	a	k	#vars	#clauses	sec	sec	sec
50	20	50	240	7955	-	104.0	-
70	20	50	350	14587	-	-	-
500	10	50	1248	42861	-	44.8	17.5
1000	15	40	3471	207353	-	239.6	81.3
3000	7	30	4630	134561	2.6	0.2	3.0
3000	9	30	5743	262653	19.0	0.3	9.0
3000	10	30	6894	278593	48.3	2.9	6.5
4000	9	30	7633	355941	11.9	0.5	15.0
4000	10	30	9011	369797	63.7	0.7	10.6
4000	11	30	9920	392125	50.4	0.5	8.6
5000	7	40	7588	223229	23.3	0.2	5.5
5000	8	30	7951	361377	8.2	0.3	18.0
5000	8	40	8197	364909	37.3	0.5	15.6
5000	10	30	11231	468605	50.9	0.7	15.9
5000	11	30	12709	509789	51.4	0.9	12.4
6000	8	30	9420	423497	11.2	0.3	23.3
6000	8	35	9496	426321	23.3	0.4	23.6
6000	9	35	11521	544685	61.8	0.9	32.3
7000	6	60	9931	181121	26.6	0.2	3.5
8000	9	30	14641	710397	12.5	0.6	54.0

Chapter 6

Attacking AES and LED

Motivation In this chapter we discuss fault attacks on scaled variants of the block ciphers AES and the lightweight cipher LED (see Sections 2.7 and 2.8). Instead of crafting fault equations manually, we introduce the tool `AutoFault` for creating automatic fault equations. To push the automatization even further, the equations are derived automatically from the hardware description of the cryptosystem. Moreover, we mention various new ways how algebraic fault attacks (AFA) can be encoded. All in all, `AutoFault` has these main features.

- `AutoFault` *is automatic.* Fault equations are derived automatically, i.e. they can be significantly more complicated than ones derived by a human cryptanalist. However, the constraints given by a "computer" may be more useful, i.e. they may restrict the search space tighter than ad-hoc fault equations.
- `AutoFault` *is user-friendly.* `AutoFault` is taylored in a way such that launching an AFA is very easy – it requires only a hardware description of the cipher. That enables us, for example, to test newly designed block ciphers in an automatic way. If the block cipher does not expose any vulnerability using `AutoFault`, it implies a certain degree of security against fault attacks.
- `AutoFault` *supports various encodings.* `AutoFault` is able to parse an algebraic description of a cryptosystem in ANF (i.e. a set of Boolean polynomials) that is given by a cryptanalyst and mix it with the automatic encoding derived from the hardware description.

Related work The standard reference for hand-crafting fault equations for AES and LED is [65, 112]. A framework for automatically creating fault equations can be found in [115]. The secret is determined by solving the set of constraints describing the cryptosystem under attack, i.e. an algebraic representation has to be created. In contrast to [115], `AutoFault` uses hardware descriptions which are usually publicly available (more details can be also found in [94]). Comparing `AutoFault` to statistical approaches, e.g. in [48, 79], a successful AFA requires less fault injections, but of higher precision. Countermeasures against fault attacks include low-level approaches like adding shields to cover metallization levels [77], placing sensors in the circuitry [95], or higher-level methods such as error-detecting codes [66]. To the best of our knowledge, `AutoFault` is the first approach to break LED-64 using no "manual" cryptanalytic information (that is, without manually derived constraints beyond the circuit or the fault descriptions). This chapter is based on [28, 29, 46].

Structure and contents In Sect. 6.1 and 6.2 we give a brief overview of the block ciphers AES, ssAES, and LED. In Sect. 6.3 we provide sufficient details how to apply a practical AFA using `AutoFault` on these ciphers. The chapter is concluded by Sect. 6.4 where a variety of AFA experiments derived by `AutoFault` under different attack assumptions are presented.

6.1 Description of AES

AES (the Advanced Encryption Standard) is a family of block ciphers based on substitution permutation networks (see Example 2.36). In the following, we focus only on its 128-bit variant, i.e. AES-128 which has 128-bit long keys and blocks and consists of 10 rounds. In this chapter we mean by AES not the whole family but only its 128-bit version. The attacks presented in the subsequent sections can be easily generalized to its other versions. In particular, ssAES (Small Scale AES) defined in [31] is a family of scaled versions of AES depending on a size of its state and the number of rounds. The small versions of AES are used for benchmarking of the attacks because they reflect the structure of the full AES.

The descriptions of the ciphers in this section are based on [36] and [31], where the complete specifications can be found. We do not discuss all modifications of the cipher. Instead, we give a brief overview of the full AES. The other variants of ssAES are similar. AES consists of 10 rounds, and its state is arranged column-wise in a 4×4 matrix of bytes $s_1, \dots, s_{16}$, i.e. the state s is represented by the matrix

$$s = \begin{bmatrix} s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \\ s_4 & s_8 & s_{12} & s_{16} \end{bmatrix}.$$

Let us mention the notation that is used for ssAES. The cipher ssAES-a-b-c-d denotes a small version of AES with the state matrix of size $a \times b$ containing c-bit words and d rounds. We write only ssAES-a-b-c if the number of the rounds is known from the context. For instance, ssAES-2-2-4 stands for AES with a state of 2×2 entries, where each entry is a 4-bit nibble. The fully-fledged AES would be written as ssAES-4-4-8-10 in this notation.

Bytes can be written in the hexadecimal numeral system as two nibbles, e.g. the hex number `0B` corresponds to the binary word `00001011`. Moreover, the bytes are considered to be the elements of the finite field $\mathbb{F}_{2^8}$ of 256 elements. As usual, individual round keys are established during a key schedule phase. The details of the key schedule of AES are not important in our context, and we refer the reader to the full specification in [36].

Next we recall the round functions used in AES.

- `SubBytes`. Each byte x of the state matrix is replaced by $S(x)$, where $S : \mathbb{F}_{2^8} \to \mathbb{F}_{2^8}$ is a non-linear function defined by $S(x) = A \cdot x^{-1} + b$ with a constant matrix A and a constant vector b, and x^{-1} denotes multiplicative inversion of $x \in \mathbb{F}_{2^8}$.

- `ShiftRows`. The ith row of the state matrix (with $i = 1, 2, 3, 4$) is shifted cyclically to the left by $i - 1$ bytes.
- `MixColumn`. Each column v of the state matrix is replaced by $M \cdot v$, where

$$M = \begin{bmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{bmatrix}.$$

- `AddRoundKey`. Each byte of the state matrix is added to the corresponding byte of the round key. The addition is done in $\mathbb{F}_{2^8}$.

6.2 Description of LED

LED is an example of the family of substitution permutation networks that is lightweight, i.e. the cipher is taylored for a usage in a limited environment. By LED we refer here to its 64-bit variant, i.e. to LED-64. For a complete specification, we refer to [49].

A state of LED has 32 rounds and is arranged row-wise in a 4×4 matrix of nibbles (i.e., of elements of $\mathbb{F}_{2^4}$). Recall that the nibbles correspond to hex numbers, e.g. `C` corresponds to the binary word `1100`. The state s is represented by the matrix

$$s = \begin{bmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ s_{12} & s_{13} & s_{14} & s_{15} \end{bmatrix}.$$

LED is optimised for hardware and is very compact. Among other features, we point out that LED has no key schedule. The round functions of LED are of the following types.

- `AddConstant`. The LED round 6-bit constant are defined for each round. From the LED round constant two nibbles `x` and `y` are derived. The following matrix A is then added to the state matrix.

$$A = \begin{bmatrix} \texttt{0} & \texttt{x} & \texttt{0} & \texttt{0} \\ \texttt{1} & \texttt{y} & \texttt{0} & \texttt{0} \\ \texttt{2} & \texttt{x} & \texttt{0} & \texttt{0} \\ \texttt{3} & \texttt{y} & \texttt{0} & \texttt{0} \end{bmatrix}.$$

- `SubCell`. Each byte x of the state matrix is replaced by $S(x)$ where $S : \mathbb{F}_{2^4} \to \mathbb{F}_{2^4}$ is a non-linear map defined by a look-up table.
- `ShiftRows`. For $i \in \{1, 2, 3, 4\}$, the ith row of the state matrix is shifted cyclically to the left by $i - 1$ bytes.

- `MixColumnsSerial`. Each column v of the state matrix is replaced by $M \cdot v$, where

$$M = \begin{bmatrix} \mathtt{4} & \mathtt{1} & \mathtt{2} & \mathtt{2} \\ \mathtt{8} & \mathtt{6} & \mathtt{5} & \mathtt{6} \\ \mathtt{B} & \mathtt{E} & \mathtt{A} & \mathtt{9} \\ \mathtt{2} & \mathtt{2} & \mathtt{F} & \mathtt{B} \end{bmatrix}.$$

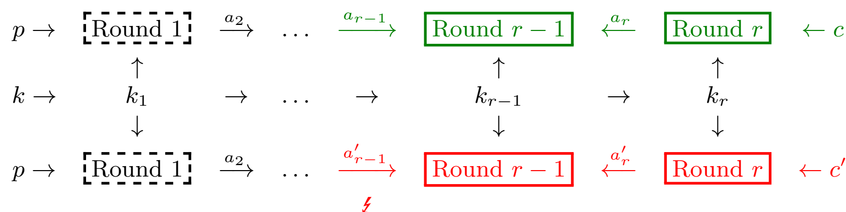
- `AddRoundKey`. Each byte of the state matrix is added to the corresponding byte of the key (in the beginning of the encryption and always after 4 rounds).

6.3 An Automatic Construction of AFA

Algebraic fault attacks on iterated block ciphers have been already described in Examples 2.41 and 2.42. In this section, we improve the framework in Example 2.42 such that attacks on AES, ssAES and LED become more powerful.

Let us start with a general improvement of the attack in Example 2.42. The improvement in the next example is inspired by the meet-in-the-middle concept.

Example 6.1. In the setting of Example 2.42, let us consider a differential fault attack (DFA) based on the following diagram. Notice the directions of the arrows in the diagram which induce a different encoding of the last rounds.



The state a_{r-1} (or the state a'_{r-1}) is propagated to the output of the round $r-1$. The output variables of the round $r-1$ are set equal to the propagation of the correct ciphertext c (or the faulty ciphertext c') backwards to a_r (or a'_r). △

The only difference between Example 2.42 and 6.1 is the "direction" of the workflow in the last rounds. The resulting algebraic description of the attack has a more convenient encoding in ANF because the substitution chain is broken down into two pieces. Thus this workflow tends to produce better CNF encodings as well.

There are three main models how to encode a part of the circuit (and consequently the whole attack such as one in Example 6.1) into CNF. In the following, we give a short overview of functional, structural, and mixed models. For each model we describe how to encode an AFA instance into CNF such that the result can be passed to a SAT solver.

Functional models The attack in the functional model is represented by a set of Boolean polynomials, and the resulting polynomials in ANF are converted to CNF by the methods explained in Sect. 4.2. A concrete encoding is given in Example 2.38.

Structural models Similar to the functional model, the structural model of an AFA consists of the last rounds of a fault-free and of a faulty encryption which are modeled as a circuit. The difference between the fault-free and the faulty encryption is achieved by adding `XOR` gates.

The functional (high-level) hardware description of the cipher (e.g., in the hardware design language `VHDL` or in higher-level behavioral languages like `SystemC`) is a prerequisite for implementing the cipher in hardware. Thus we suppose that the description is readily available. The round and key schedule information are automatically extracted from the `VHDL` files and combined into a new `VHDL` file which contains only the parts of the cipher required for the fault attack.

The new `VHDL` file which contains only the essential parts of the cipher is then synthesized to a gate-level hardware description. For this purpose we may use a design compiler (such as the `Synopsis Design Compiler`), a standard logic-synthesis tool broadly used by industry.

The transformation to CNF is done by the *Tseitin transformation* (e.g. see [111]). The Tseitin transformation is applicable in our setting because the model is a combinational circuit, i.e. a circuit composed of logic gates with no memory. The Tseitin transformation introduces a new variable for each internal line in the circuit and maps each logic gate onto a set of clauses implementing its characteristic function. A simple application of the Tseitin transformation is given in the next example.

Example 6.2. Consider a circuit consisting of one `AND` gate with Boolean inputs X_1, X_2 and one output X_3. The characteristic function of the circuit is given by $X_3 \equiv (X_1 \wedge X_2)$, which corresponds to the set of clauses $\big\{\{\bar{X}_3, X_1\}, \{\bar{X}_3, X_2\}, \{\bar{X}_1, \bar{X}_2, X_3\}\big\}$ via the Tseitin transformation. △

Experiences from related fields such as SAT-based automatic test pattern generation (for instance, see [15]) show that clauses generated by the Tseitin transformation significantly reduce the solving time compared to a direct translation of a circuit to a Boolean formula (e.g., by using term rewriting).

Mixed models Mixed models are obtained by combining functional and structural models, i.e. the attack is divided into two (not necessary disjunct) parts, and each part is encoded either in the functional or in the structural model. The underlying idea regarding combining two models is to give the SAT solver additional information and guide it towards a satisfying solution faster.

Since the functional and the structural models are both sufficient to model the entire fault attack on their own, one can also add only part of one model to the other one. To stress this out, we call such models *partially mixed models*.

6.4 Experiments

To evaluate structural and mixed functional-structural models, we focused on attacking the medium-sized ssAES-2-2-4. Table 6.1 summarizes the size of the obtained fault attack models in CNF by the different methods. Whereas the sparse representation uses

the fewest variables, the other representations add new helper variables which, in turn, decrease the number of clauses (see Sect. 4.2). For the initial experiments, we assume that the fault occurs in the first nibble of the state, and that only one bit is faulty.

Table 6.1: Sizes of the formulae for the DFA models on ssAES-2-2-4

Models	# Clauses	# Variables
Functional (sparse representation)	24 515	288
Functional (dense representation)	13 936	1272
Structural	3 086	916

For experiments, the SAT solver `antom` [101] is used on a single core of an Intel Xeon E5-2643 CPU clocked at 3.3 GHz. To compare the different attack models, we created 250 different random fault attacks instances and measured the time required from the beginning of the solving process until the correct key was found. Figure 6.1 shows the average runtime as well as the number of key candidates before the solution was found.

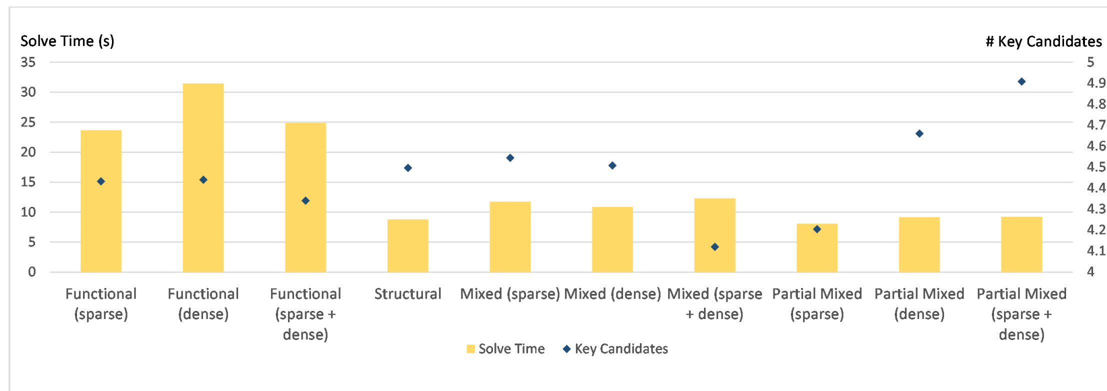


Figure 6.1: The solving time and the number of key candidates for several DFA models on ssAES-2-2-4

It turns out that the structural fault attack model performs better. E.g. compare the average solving time with the timings for the sparse or the dense encodings. Note that the number of clauses in itself does not play an important role (e.g. compare the results for the dense and the sparse encodings).

Combinations of structural with functional models are a little slower than the vanilla structural model, but always faster than the functional model alone. When combining the structural model with some clauses from the functional model into a partially mixed model, the solving speed is slightly increased in comparison to the complete mixed model. However, some partially mixed models give us sometimes the fastest solving speed.

The average number of key candidates (until the correct key is found) in our experiments is between 4 and 5. However, there seems to be no correlation between the number of key candidates and the solving time. Recall that the SAT solver traverses the solution search space in a very complex manner, and it may be faster to discover

Table 6.2: The size of CNF formulae corresponding to AFA.

Cipher	Block Size	# Rounds	Formula size (# Clauses)
ssAES-2-2-4	16 bits	10	3 086
ssAES-4-4-4	64 bits	10	13 420
LED-64	64 bits	32	15 544

inconsistent key candidates than the correct key.

Based on later experiments, we decided to choose the structural model of AFA, and we perform some more tests with ssAES and LED. Moreover, we tested AutoFault with different fault models. AutoFault is applied to three different ciphers: two variants of ssAES and the 64-bit version of the block cipher LED. Table 6.2 gives an overview of the ciphers as well as the size of the CNF formulas corresponding to the attacks.

In the case of ssAES and LED, fault injections in different locations were simulated in software. Consistent with work in [29], we consider faults in one nibble and in two neighboring nibbles three rounds before the termination of encryption. For our experiments, we require that either exactly one, exactly two, or up to four or eight bits are flipped. (The last two scenarios correspond to "nibble faults" and "byte faults" in [64, 116].)

Additionally, for the LED cipher we also consider two more advanced fault injections to highlight the versatility of our approach. Firstly, a single bit is flipped anywhere within the 64-bit input data. This could correspond to a timing-based attack where the attacker increases the clock frequency (or decreases the supply voltage) until the output of the cipher is incorrect. Note that the attacker may not actually know which bit is affected. Secondly, we consider an attack where the entire first (or the first and the second nibble) is corrupted (with up to 4 or 8 faulty bits, respectively). This corresponds to a physical attack where the location of the fault can be narrowly controlled to a small area, but the number of flipped bits is random. For these attacks, we randomly set the number of actual faults to anywhere between 1 and the maximum number possible for each case.

The averaged results for up to 10,000 automatically generated attacks (with an overall timeout of 40 hours for each experiment) are shown in Table 6.3. All experiments were run on an Intel Xeon E5-2643 CPU clocked at 3.3 GHz. Moreover, the distributions of the solving times for the three ciphers are reported in Figures 6.2a, 6.2b and 6.3 (the minimum and the maximum solving times are indicated by the lines and the 50% interquartile range are indicated by rectangles). Note that AutoFault terminates the attack when the found key candidate was confirmed to be the correct key by simulation, that is, the attacks indeed break the cipher.

Notice that AutoFault is able to find valid attacks for the considered ciphers, despite the fact that it incorporates no cipher-specific cryptanalysis. This might be expected in the case of ssAES, which has been designed as a synthetic benchmark for cryptanalysis and is not recommended for use in actual cryptographic applications. However, it is remarkable that a fully-fledged (though lightweight) cipher such as LED has so little

Table 6.3: The solving times and the numbers key candidates for ssAES-2-2-4, ssAES-4-4-4 and LED-64

Cipher	Fault model	Mean solve time (sec)	Average number of key candidates
ssAES-2-2-4	1 bit, 1st nibble	16.46	5.16
	1 bit, 1st/2nd nibble	16.39	7.61
	2 bits, 1st nibble	16.32	11.93
	2 bits, 1st/2nd nibble	17.98	25.76
ssAES-4-4-4	1 bit, 1st nibble	9 574.61	620.26
	1 bit, 1st/2nd nibble	7 173.82	324.18
	2 bits, 1st nibble	26 357.30	170.40
	2 bits, 1st/2nd nibble	23 651.00	55.00
LED-64	1 bit, 1st nibble	254.78	3 508.33
	1 bit, 1st/2nd nibble	442.72	3 044.23
	2 bits, 1st nibble	384.96	6 395.38
	2 bits, 1st/2nd nibble	847.77	2 303.87
	1 bit, any nibble	712.70	4 896.29
	1 bit, after S-Box	127.66	6 858.65
	≤ 4 bits, 1st nibble	365.78	7 051.83
	≤ 8 bits, 1st/2nd nibble	762.79	1 163.36

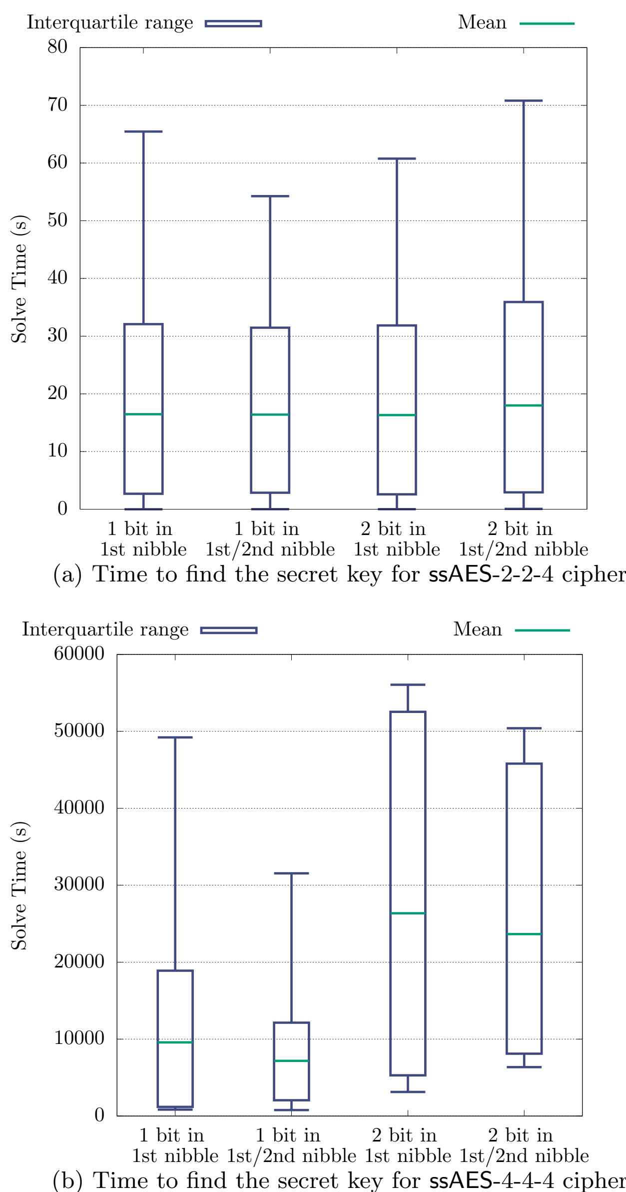
resistance against `AutoFault`.

When comparing our results on LED-64 with earlier attacks on the same cipher, one notices that the typical number of around 7,000 considered key candidates is inconsistent with the much larger key candidate space size of $2^{19} - 2^{26}$ reported in [65], even though our DFA setting is essentially identical to the construction in [65]. This can be attributed to two factors. Firstly, the key candidates in [65] may include inconsistent candidates as well. Therefore, the actual number of consistent candidates may be lower than $2^{19} - 2^{26}$. Secondly, we used the SAT solver in the incremental mode, i.e. it is possible that the solver infers further information from conflict clauses that exclude not only one but several invalid key candidates at once. If this happens, the attack needs less than one iteration per key candidate, in contrast to the conventional brute-force search.

With regard to the run time, it is not surprising that the necessary effort increases significantly for larger states of an otherwise identical cipher (e.g. ssAES-4-4-4 vs. ssAES-2-2-4). This is expected, and it is worth noting that the attack does not yet terminate before timeout for the fully-fledged AES, i.e. for ssAES-4-4-8. However, the results for ssAES is useful because they illustrate scalability as a function of the fault model.

The execution times for LED-64 via `AutoFault` are better than the several hours reported in [64], but worse than the timings given in [116]. Compared to [64], `AutoFault` uses a more compact differential models where the attack-unaffected cipher parts are skipped, and also employs a more efficient CNF formula construction out of com-

binational circuitry (rather than from the functional description of the cipher). Zhao et al. [116] employed special constructions (with reverse models of cipher rounds) and added clauses derived from fault-affected differentials to facilitate the search. Approximately one order of magnitude slowdown appears to be adequate for a fully-automatic attack construction without any cipher-specific analysis.



(a) Time to find the secret key for ssAES-2-2-4 cipher

(b) Time to find the secret key for ssAES-4-4-4 cipher

Figure 6.2: Time to find the secret key for the ssAES cipher

The work on the tool `AutoFault` is still ongoing. Using the latest parallel SAT solvers in `AutoFault`, it is possible to break full-scale AES (e.g. using 2 fault injections) or other

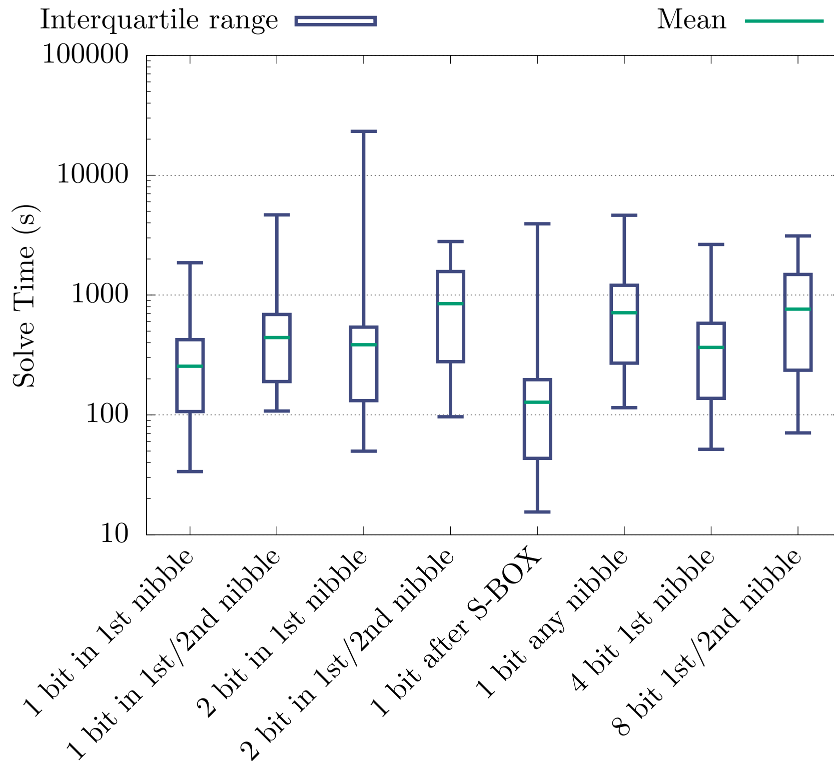


Figure 6.3: Time to find the secret key for the LED-64 cipher w.r.t. the injections in the various locations

state-of-the-art ciphers such as PRESENT defined in [19] (e.g. using 5 fault injections)[1].

[1]These results follow from an yet unpublished article of M. Gay et al. which was submitted to FDTC 2019.

Chapter 7

Attacking SHA using Programmatic SAT Solvers

Motivation In this chapter we introduce a new approach to algebraic fault attacks (AFA) using programmatic SAT solvers (see Sect. 2.5 and 2.8). The programmatic interface is used to strengthen the performance of the base solver on the PREIMAGE problem (defined in Sect. 2.7) of the SHA families. While encoding SHA into CNF in the similar manner as in [91], we observed that the resulting CNF does not perform well with the respect to the native Boolean Constraint Propagation (BCP, see Algorithm 2.4) of the SAT solver `MapleSAT` [80]. This problem is captured by the notion of a special version of general arc consistency (GAC).

There are two straightforward solutions to this problem – either we change the CNF encoding, or we modify the propagation mechanism. On one hand, the latter solution is not easily applied in the base solver because the BCP architecture is deeply ingrained in modern SAT solvers. On the other hand, changing the CNF encoding such that it maintains GAC results usually in a larger CNF formula. However, the propagation can be easily enhanced via the programmatic interface. Moreover, the programmatic interface can implement other features of the attack (e.g. checking of message candidates, etc.) that has to be taken care of otherwise from the "outside" of the solver.

Related work The initial work on fault attacks on the SHA family goes back as far as [78], where a differential fault attack (DFA) is applied on SHACAL-1 (i.e. a block cipher adopting the structure of SHA-1). Authors of [51] extended the attack to SHA-1. Because the structure of SHA-1 is more difficult than SHACAL-1, they needed more than a thousand faults to derive a message. A fault attack on the HMAC setting of SHA-2 is proposed in [62]. There it is shown that message values of size n can be recovered with approximately $n/3$ faults. An AFA on SHA-2 using SMT methods, namely the constraint solver for the theory of quantifier-free bit-vectors `STP`, is presented in [50]. (For details on `STP`, see [43].) Our attack via the programmatic interface presented in this chapter outperforms the other approaches in terms of the number of fault injections and solving time. This chapter is based on [84].

Structure and contents In Sect. 7.1 we recall some parts from the theory of constraint satisfaction problems. In Sections 7.2 and 7.3 descriptions of SHA-1 and SHA-2 hash functions are given. In Sect. 7.4 we describe the inversion attack using programmatic solvers. Namely, we explain its two basic components: the programmatic propagator and the programmatic conflict analyzer. Experiments and results are given in Sect. 7.5.

7.1 Preliminaries

The following definition is motivated by the notion of arc consistency in the field of constraint satisfaction problems (CSP, see Sect. 2.6). We spell out a modified version of *general arc consistency* (GAC) given in [7].

Definition 7.1. Let C be a set of clauses encoding a Boolean map φ, and let ri be an inference rule of propositional logic. We say that C ri-**maintains input/output GAC** if for any assignment α of the variables corresponding to the input variables of φ, the assignment of the variables corresponding to the output variables of φ can be derived from $C|_\alpha$ by applying ri.

In order to use the "full" strength of a solver based on a rule of inference, we have to choose an encoding that maintains input/output GAC. The following example illustrates the core of the problem of the encodings that do not maintain GAC under unit propagation. (Recall that unit propagation (UP) is the default propagation procedure in SAT solvers.)

Example 7.2. Consider the pseudo-Boolean constraint $x_1 + x_2 \leq 0$ with $x_1, x_2 \in \{0, 1\}$ and integer addition "+". We can encode this constraint into a CNF formula C by using a half-adder with inputs x_1 and x_2 and by forcing the outputs to be zero. The half-adder relations for the carry bit A and the sum bit B can be described as $A \leftrightarrow X_1 \wedge X_2$ and $B \leftrightarrow X_1 \oplus X_2$. The final encoding of the formula

$$(A \leftrightarrow X_1 \wedge X_2) \wedge (A \leftrightarrow X_1 \oplus X_2) \wedge (\bar{A} \wedge \bar{B})$$

into CNF is equal to $C = \big\{\{\bar{X}_1, X_2\}, \{X_1, \bar{X}_2)\}, \{\bar{X}_1, \bar{X}_2\}\big\}$. It is clear that X_1 and X_2 should be set to zero. But these values are not discovered by applying BCP on C. $\triangle$

7.2 Description of SHA-1

SHA-1 was designed by NSA and standardized by NIST in 1995 (see the standard in [88]). It was widely used in many applications, but after successful collision attacks reported in [107] [1], security practitioners moved away to stronger alternatives such as SHA-2 or SHA-3. However, SHA-1 is still considered resistant against preimage and second preimage attacks.

SHA-1 is an iterated hash function family based on the Merkle-Damgård construction (see Example 2.37), where each message block has 512 bits. Each block is given to the SHA-1 compression function that outputs 160 bits. We recall only a part of the SHA-1 specification. For the full description of SHA-1, we refer to [88].

The *message expansion* relation for expanding 16 initial message 32-bit words $w_0, \ldots,$ w_{15} to 80 32-bit words $w_0, \ldots, w_{79}$ is defined by

$$w_i = (w_{i-3} \oplus w_{i-8} \oplus w_{i-14} \oplus w_{i-16}) \lll 1, \text{ for } i \in \{16, \ldots, 79\}. \qquad \text{(I-M)}$$

[1] A practical collision attack on SHA-1 can be found at `https://shattered.io/`

where "$\lll i$" denotes left rotation by $i \in \mathbb{N}$ positions. The internal state of SHA-1 has 160 bits, and it is divided in five 32-bit words $a_i, \ldots, e_i$ for each iteration $i \in \{0, \ldots, 79\}$. The *updating function* for the iteration $i = 0, \ldots, 79$ is defined as follows

$$(a_{i+1},\, b_{i+1},\, c_{i+1},\, d_{i+1},\, e_{i+1}) \leftarrow \big(F_i(b_i, c_i, d_i) \boxplus e_i \boxplus (a_i \lll 5) \boxplus w_i \boxplus k_i,\, a_i,\, b_i \lll 30,\, c_i,\, d_i\big), \quad \text{(I-U)}$$

where "$\boxplus$" denotes integer addition modulo 2^{32} and k_i is an iteration-specific constant. The Boolean map $F_i : \mathbb{F}_2^{32} \times \mathbb{F}_2^{32} \times \mathbb{F}_2^{32} \to \mathbb{F}_2^{32}$ changes after every 20 rounds and is one of the following functions. (The operations, like "$\wedge$" or "$^-$", are applied bit-wise.)

- $\texttt{Ch}(x, y, z) = (x \wedge y) \oplus (\bar{x} \wedge z)$
- $\texttt{Xor}(x, y, z) = x \oplus y \oplus z$
- $\texttt{Maj}(x, y, z) = (x \wedge y) \oplus (y \wedge z) \oplus (x \wedge z)$

7.3 Description of SHA-256

In this section we briefly describe the standard hash function family of SHA-2 [38]. More precisely, we focus only on SHA-256. (In the following, we use SHA-2 and SHA-256 interchangeably.) The structure of SHA-2 is similar to SHA-1 (i.e. SHA-2 is an iterated hash function family as in Example 2.37), but its updating function and its message expansion procedure are more complex. The input block size is 512 bits and it has 64 iterations. Using the following *message expansion* relation, 16 32-bit input words $w_0, \ldots, w_{15}$ are expanded to 64 32-bit words $w_0, \ldots, w_{63}$.

$$w_i = \sigma_1(w_{i-2}) \boxplus w_{i-7} \boxplus \sigma_0(w_{i-15}) \boxplus w_{i-16}, \text{ for } i \in \{16, \ldots, 63\}, \quad \text{(II-M)}$$

where σ_0 and σ_1 are Boolean maps $\mathbb{F}_2^{32} \to \mathbb{F}_2^{32}$ defined as follows.

$$\begin{aligned} \sigma_0(x) &= (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3), \\ \sigma_1(x) &= (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10), \end{aligned}$$

where "$\gg i$" denotes rotation to the right by $i \in \mathbb{N}$ positions. The internal state of SHA-2 has 256 bits consisting of eight 32-bit words labelled as $a_i, b_i, \ldots, h_i$ for each iteration i. The *updating function* is defined as follows.

$$(a_{i+1},\, b_{i+1},\, c_{i+1},\, d_{i+1},\, e_{i+1},\, f_{i+1},\, g_{i+1},\, h_{i+1}) \leftarrow (t_1 \boxplus t_2,\, a_i,\, b_i,\, c_i,\, d_i \boxplus t_1,\, e_i,\, f_i,\, g_i) \quad \text{(II-U)}$$

with

$$\begin{aligned} t_1 &= h_i \boxplus \Sigma_1(e_i) \boxplus \texttt{Ch}(e_i, f_i, g_i) \boxplus k_i \boxplus w_i, \\ t_2 &= \Sigma_0(a_i) \boxplus \texttt{Maj}(a_i, b_i, c_i), \\ \Sigma_0(x) &= (x \ggg 2) \oplus (x \ggg 13) \oplus (x \ggg 22), \\ \Sigma_1(x) &= (x \ggg 6) \oplus (x \ggg 11) \oplus (x \ggg 25), \end{aligned}$$

where the functions Ch, Maj, and $\boxplus$ are the same as for SHA-1, k_i denotes the SHA-2 constants, and w_i denotes the processed expanded message block.

7.4 Algebraic Fault Attacks on SHA

In this section we propose a programmatic SAT solver-based approach to algebraic fault attacks. As our target we choose the problem PREIMAGE (defined in Sect. 2.7) of the SHA-1 and the SHA-2 family. However, the method general enough such that it can be applied in other primitives under various cryptanalytic scenarios.

An instance of the problem PREIMAGE is encoded in CNF as well as in a programmatic interface of a SAT solver. Using the programmatic encoding, we extend the Boolean constraint propagator (BCP) to a programmatic propagator and the conflict analysis extension to a programmatic conflict analyzer.

- *Programmatic propagator.* Given a Boolean map φ over the input variables x and the output variables y, there sometimes exists a set of clauses C_φ which encodes φ such that the standard BCP does not propagate the values assigned to x all the way to y, i.e. C_φ does not UP-maintain input/output GAC (see Def. 7.1). In this case, the programmatic propagator directly sets the bits corresponding to y by calculating $\varphi(x)$ in the programmatic interface.
- *Programmatic conflict analyzer.* The programmatic conflict analyzer verifies whether a candidate for the solution that is found by the base solver is valid, i.e. if it satisfies all the constraints. If the solution is not valid, a conflict clause is added to the conflict clause database of the solver. Otherwise, the validated solution is returned by the solver. The analyzer is called in the inner loop of the SAT solver, and thus it takes advantage of its inherent incrementality.

After a high-level description of the individual components, we focus on details regarding the attack on the SHA family. The SHA compression functions use multi-operand additions. The encoding of the adders in [90] gives a very compact CNF. However, the BCP is not very efficient for this type of encodings (see Example 7.2). Namely, it does not UP-maintain input/output GAC. That is why we implement a SHA *propagator* that strengthens the BCP. The programmatic propagation is called in the main search loop of the solver after BCP is done, and no conflicts are detected. The callback function then checks whether there exist any other bits that could be set. In the next example we illustrate how it is done. (Note that the example is rather artificial. In practise, we deal with larger Boolean functions.)

Example 7.3. Consider the Boolean function $\varphi : \mathbb{F}_2 \times \mathbb{F}_2 \to \mathbb{F}_2$ given by $(x_1, x_2) \mapsto x_3 := x_1 \oplus x_2$. Assume that a set of clauses C which encodes φ is given to a SAT solver that assigns $X_1 \mapsto \texttt{True}$ and $X_2 \mapsto \texttt{False}$. If X_3 has not been set, the programmatic propagator adds a clause $\{\bar{X}_1, X_2, X_3\}$ to C that encodes the implication $X_1 \wedge \bar{X}_2 \to X_3$. The implication forces the solver to set the output bits in the next cycle. △

Let us look closer at the SHA *conflict analyzer*. The SHA conflict analysis is invoked when the BCP reaches a state where all the variables corresponding to the message are assigned, and there is no conflict. The analyzer identifies the message bits and checks that the message hashes to the given target. If not, a conflict clause that blocks the

current spurious message bits is returned to the solver. Because the solver has the reason clauses that led to the partial assignment, it can further optimize learned clauses.

In the following we describe how to recover the last chaining value by a fault attack. Even though the setting of the attack seems to be a bit artificial (in the sense that there is no secret key stored in the implementation of the hash function), the attack is a foundation stone for a forgery attack on message authentication codes HMAC created from SHA. (See [50, Sect. 4] for details on the attack, see [108, Sect. 4.4.] for details on HMACs). The number $w_H(s)$ denotes the Hamming weight of a string s.

In the attack described in Algorithm 7.1, we target the last 16 rounds of the SHA-1 compression function. The message expansion is invertible, provided we have 16 consecutive words (see Equation (I-M)). This means that recovering the last 16 expanded message words enables us to recover all message bits. Therefore we inject faults to the input of the last 16 rounds, and more particularly in b_{64}. This fault location is more desirable because of the way the fault propagates in the next rounds. For more details we refer to [51].

Let f be the compression function of SHA-1. Let $f_{1..64}$ (resp. $f_{65..82}$) be the Boolean map representing the first 64 rounds of f (resp. the last 16 rounds of f). Thus we have the following composition $f = f_{65..80} \circ f_{1..64}$. In Step 4 we inject the fault δ_i. Step 7 encodes the instance of the algebraic attack C together with the fault equations C_i. Steps 8–16 corresponds to an integration of the SAT solver with the SHA propagator and the SHA conflict analyzer.

Rewriting Algorithm 7.1 into the setting of SHA-2, we get an attack on SHA-2 in Algorithm 7.2. (Algorithm 7.2 uses the same notation introduced in Algorithm 7.1.) We first target the last four rounds. Thus we get a set of candidates for $(w_{61}, \ldots, w_{64})$. Then we target the last 8 rounds in order to get a set of candidates for $(w_{57}, \ldots, w_{60})$. The injection procedure is repeated in round 52 and 48 such that we get enough information about $w_{65}, \ldots, w_{80}$. This multi-stage attack is inspired by the approach in [50].

7.5 Experiments

On one hand, there are SAT solvers such as `CryptoMiniSAT` [106] that implement a `XOR` reasoning which could be beneficial in solving cryptographic problems based on SHA-1 and SHA-2. On the other hand, SMT solvers that handle bit-vectors such as `STP` [43] are good candidates for solving these kinds of instances as well. However, according to the results in [85], `MapleSAT` [80] outperforms all of them on SHA-1 preimage instances.

Because the SHA-1 preimage instances are similar to the premimage algebraic fault instances, we decided to use `MapleSAT` to implement the programmatic callbacks in C++. (We also experimented with `Opturion CPX` [92], which is a constraint solver known from the Minizinc challenge in 2015 that combines CSP and SAT solving techniques. It has turned out that `Opturion CPX` could solve only a small number of our AFA instances.)

Furthermore, we have also decided to use the multi-armed bandit restart (MABR) policy described in [85] for `MapleSAT`, which adds a performance gain on cryptographic instances. We run tests under various assumptions on the number of the injected faults and on the maximal weight of the faults. For each experiment we generated 100 random

Algorithm 7.1 `AFA-SHA-1` (An AFA on SHA-1)

Input: The compression function f of SHA-1, the number of faults $k \in \mathbb{N}$, the maximal weight of faults $d \in \mathbb{N}$, a 160-bit digest h such that there exists a 512-bit message m with $f(m) = h$.

Output: A 512-bit word m', such that $f(m') = h$.

1: Let C be a CNF encoding of $f_{65..80}(x) = h$.
2: **for** $i = 1, \dots, k$ **do**
3: Generate a random fault value δ_i with $w_H(\delta_i) \leq d$.
4: $h_i := f_{65..80}(f_{1..64}(m) \oplus \delta_i)$
5: Let C_i be a CNF encoding of $h_i = f_{65..80}(x \oplus \delta_i)$.
6: **end for**
7: $Q := C \wedge \bigwedge C_i$
8: **repeat**
9: Find a model α for Q.
10: Extract the assignment for $w_{65}, \dots, w_{80}$ from α.
11: **for** $j = 64, \dots, 1$ **do**
12: $w_j := (w_{j+16} \ggg 1) \oplus w_{j+13} \oplus w_{j+8} \oplus w_{j+2}$
13: **end for**
14: $m' := w_0 \| \dots \| w_{15}$
15: $Q := Q \wedge c$, where c denotes a clause that blocks the assignment corresponding to m'.
16: **until** $f(m') = h$
17: **return** m'

message-target pairs. The timeout was set to 4 hours for SHA-1 instances and 12 hours for SHA-2 instances. All experiments mentioned in these sections were executed on Intel Xeon CPUs at 3.2 GHz with 16 GB of RAM.

Table 7.1 shows the results of applying AFA on SHA-1 and SHA-2. Its rows correspond to the maximal weight of the injected faults. Its columns correspond to the number of injected faults during the attack. Starting from a single bit, going to a nibble, a single byte, single word to the 32-bit random fault model. Each element in Table 7.1 represents the number of instances (out of 100 randomly generated AFA instances) that our solver was able to solve within the time limit.

From Table 7.2a we can see that we are able to compute the message bits with as few as 8 faults in the single byte fault model. In previous attacks on SHA-1, Hemme et al. [51], apply a DFA that uses 1002 faults. In the same fault model as in [51] (i.e. the 32-bit fault model), we use only 11 faults.

Moreover, we were able to compute the SHA-2 target bits using 32 faults in the 24-bit fault model (see Table 7.2b). While Hao et al. [50] use 65 faults in the 32-bit random fault model, our method is able to find the message by injecting 48 faults in the same fault model.

Next we take a look at the performance of the programmatic AFA solver on solving the SHA algebraic fault instances. Figure 7.1 depicts the cactus plot of the vanilla `MapleSAT`

Algorithm 7.2 `AFA-SHA-2` (An AFA on SHA-2)

Input: The compression function f of SHA-2, the number of faults k such that k is divisible by 4, the maximal weight of faults d, a 256-bit digest h such that there exists a 512-bit message m such that $f(m) = h$.

Output: A 512-bit message m', such that $f(m') = h$.

1: Let C be a CNF encoding of $f_{49..64}(x) = h$.
2: $\Phi := \varphi$
3: **for** $j \in \{60, 56, 52, 48\}$ **do**
4:    **for** $i = 1, \ldots, k/4$ **do**
5:       Generate a random fault value δ_i with $w_H(\delta_i) \leq d$.
6:       $h_i := f_{(j+1)..64}(f_{1..j}(m) \oplus \delta_i)$
7:       Let C_i be a CNF encoding of $h_i = f_{(j+1)..64}(x \oplus \delta_i)$.
8:    **end for**
9: **end for**
10: $Q := C \wedge \bigwedge C_i$.
11: **repeat**
12:    Find a model α for Q.
13:    Extract the assignment for $w_{49}, \ldots, w_{64}$ from α.
14:    **for** $j = 48, \ldots, 1$ **do**
15:       $w_j := w_{j+16} \boxminus \sigma_1(w_{j+14}) \boxminus w_{j+9} \boxminus \sigma_0(w_{j+1})$, where "$\boxminus$" denotes subtraction modulo 2^{32}.
16:    **end for**
17:    $m' := w_0 \| \ldots \| w_{15}$
18:    $Q := Q \wedge c$, where c denotes a clause that blocks the assignment corresponding to m'.
19: **until** $f(m') = h$
20: **return** m'

solver and the `MapleSAT` solver with various extensions in the programmatic interface. We have turned each of the programmatic functionality on and off to see which of them contributes more to the performance of the solver. There are four solvers compared in the plot. The base version of `MapleSAT`, `MapleSAT` with the SHA propagator, `MapleSAT` with SHA conflict analyzer, and `MapleSAT` with both of these callbacks.

The timings in Figure 7.1 corresponds to the 32-bit fault model with 11 faults injected in the case of SHA-1 and 48 faults in the case of SHA-2. The plot shows that the SHA conflict analyzer can solve two more instances in SHA-1 and 14 more instances in SHA-2. However, the main performance boost is caused by the SHA programmatic propagation, which can solve 6 more instances in the case of SHA-1, and 28 more instances in the case of SHA-2.

Let us compare the total timings for solving all the instances between `MapleSAT` and fully programmatic `MapleSAT`. If we set the runtime of the timed-out instances to the time limit, we can see a 2.48x speedup for SHA-1 and 7.73x speedup for SHA-2. If we use the PAR-2 method (i.e. penalizing the timed out instances by setting their runtime

Table 7.1: The number of solved AFA instances out of 100 for different number of faults and maximal weight of the faults

Fault weight \ Number of faults	8	11	12	16	20
1	65	69	70	64	43
2	85	82	82	73	61
4	95	95	94	87	72
8	100	100	100	91	86
16	90	100	100	90	80
32	75	100	100	89	75

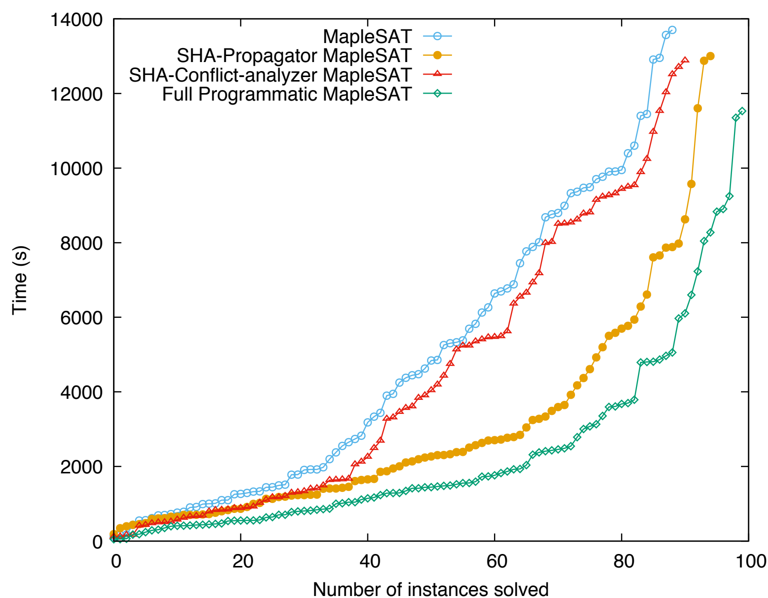
(a) SHA-1

Fault weight \ Number of faults	32	40	48	56
8	28	20	8	0
12	32	21	8	2
16	69	60	28	9
20	90	75	31	10
24	100	95	72	20
28	95	71	70	34
32	71	82	100	48

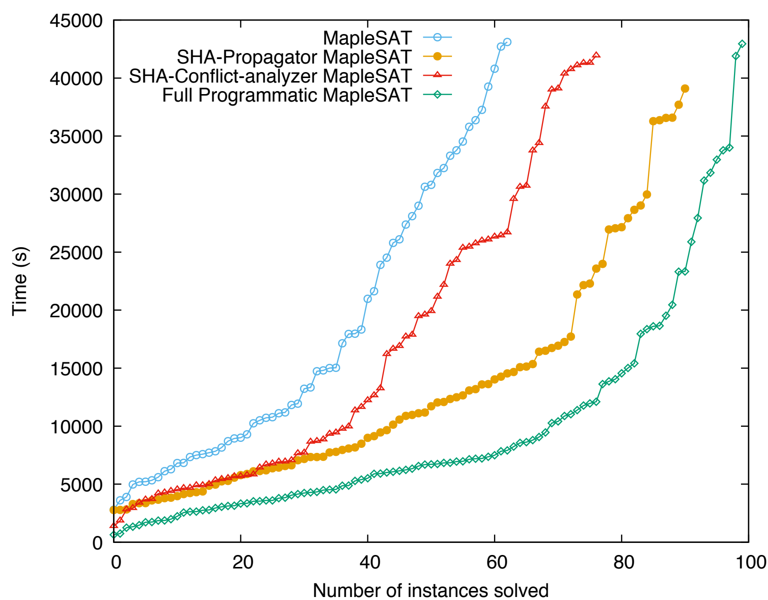
(b) SHA-2

to double the time limit), we see a 3.16x speedup in SHA-1 and 14.3x speedup in SHA-2.

Our results show the versatility of the programmatic SAT solver architecture in AFA. By taking a state-of-the-art SAT solver, we are able to extend it with programmatic functionality such that its performance is improved. Observing Table 7.1, we sometimes see that more injected faults does not imply that more instances are solved. At a first sight it seems to be counterintuitive, because adding more faults helps restrict the search space. However, adding a new fault equation, the number of clauses in the input grows rapidly (especially in the case of SHA-2), which may crucially slow down the propagation. Thus there is a trade-off between pruning the search space by new fault equations and the size of the resulting formula.



(a) 32-bit fault model AFA on SHA-1



(b) 32-bit fault model AFA on SHA-2

Figure 7.1: Cactus plots comparing `MapleSAT` with the `MapleSAT` after adding each of the programmatic callbacks. Each data point $(x, y) \in \mathbb{N} \times \mathbb{Q}$ on this plot means that x algebraic fault instances are solved in under y seconds.

Bibliography

[1] J. Abbott, A. Bigatti, and G. Lagorio. CoCoA-5: a system for doing Computations in Commutative Algebra. `http://cocoa.dima.unige.it`.

[2] M. R. Albrecht, C. Cid, J.-C. Faugère, and L. Perret. On the relation between the mxl family of algorithms and Gröbner basis algorithms. *J. Symb. Comput.*, 47, pp. 926–941, 2012.

[3] S. Arora and B. Barak. *Computational complexity: a modern approach.* Cambridge University Press, Cambridge, 2009.

[4] Y. Asahiro, K. Iwama, and E. Miyano. Random generation of test instances with controlled attributes. In D. Johnson and M. Trick (Eds.), *Cliques, Coloring, and Satisfiability*, DIMACS Series, vol. 26, pp. 377–394. American Mathematical Society, 1996.

[5] G. Audemard, G. Katsirelos, and L. Simon. A restriction of extended resolution for clause learning SAT solvers. In M. Fox and D. Poole (Eds.), *AAAI Conference on Artificial Intelligence - AAAI'10*, AAAI Press, Atlanta, 2010.

[6] G. Audemard and L. Simon. Glucose in the SAT 2014 Competition. In *SAT Competition 2014: Solver and Benchmark Descriptions*, pp. 31–32, Univ. of Helsinki, Helsinki, 2014.

[7] O. Bailleux, Y. Boufkhad, and O. Roussel. New encodings of pseudo-boolean constraints into CNF. In O. Kullmann (Ed.), *International Conference on Theory and Applications of Satisfiability Testing*, LNCS 5584, pp. 181–194, Springer-Verlag, Berlin, 2009.

[8] T. Balyo, M. J. Heule, M. Järvisalo, et al. *Proceedings of SAT Competition 2016.* Department of Computer Science, University of Helsinki, Helsinki, 2016.

[9] T. Balyo, M. J. Heule, M. Järvisalo, et al. *Proceedings of SAT Competition 2017.* Department of Computer Science, University of Helsinki, Helsinki, 2017.

[10] G. Bard. *Algebraic Cryptanalysis.* Springer-Verlag, Heidelberg, 2009.

[11] G. V. Bard, N. T. Courtois, and C. Jefferson. Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over GF(2) via SAT-solvers. Cryptology ePrint Archive, Report 2007/024, 2007. `http://eprint.iacr.org/2007/024`.

[12] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. In *Proceedings of the IEEE*, vol. 100, pp. 3056–3076. IEEE Computer Society, 2012.

[13] P. Baumgartner and F. Massacci. The taming of the (X)OR. In J. Lloyd, V. Dahl, U. Furbach, et al. (Eds.), *Computational Logic - CL 2000*, LNCS 1861, pp. 508–522, Springer-Verlag, Berlin, 2000.

[14] J. Bebel and H. Yuen. Hard SAT instances based on factoring. In A. Balint, A. Belov, M. J. Heule, et al. (Eds.), *SAT Competition 2013: Solver and Benchmark Descriptions*, pp. 102, Univ. of Helsinki, Helsinki, 2013. `https://toughsat.appspot.com/`.

[15] B. Becker, R. Drechsler, S. Eggersglüß, and M. Sauer. Recent advances in SAT-based ATPG: non-standard fault models, multi constraints and optimization. In *IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era - DTIS'14*, pp. 1–10. IEEE Computer Society, 2014. `https://ieeexplore.ieee.org/document/6850674`.

[16] A. Biere and A. Fröhlich. Evaluating CDCL variable scoring schemes. In M. Heule and S. Weaver (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2015*, LNCS 9340, pp. 405–422, Springer-Verlag, Cham, 2015.

[17] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability*. IOS Press, Amsterdam, 2009.

[18] E. Biham, R. Anderson, and L. Knudsen. Serpent: A new block cipher proposal. In S. Vaudenay (Ed.), *Fast Software Encryption*, LNCS 1372, pp. 222–238, Springer-Verlag, Berlin, 1998.

[19] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In P. Paillier and I. Verbauwhede (Eds.), *Cryptographic Hardware and Embedded Systems - CHES 2007*, LNCS 4727, pp. 450–466, Springer-Verlag, Berlin, 2007.

[20] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symb. Comput.*, 24, pp. 235–265, 1997.

[21] C. Bouillaguet, H.-C. Chen, C.-M. Cheng, T. Chou, R. Niederhagen, A. Shamir, and B.-Y. Yang. Fast exhaustive search for polynomial systems in $\mathbb{F}_2$. In S. Mangard and F.-X. Standaert (Eds.), *Cryptographic Hardware and Embedded Systems - CHES 2010*, LNCS 6225, pp. 203–218, Springer-Verlag, Berlin, 2010.

[22] M. Brickenstein. *Boolean Gröbner Bases: Theory, Algorithms and Applications*. Logos Verlag, Berlin, 2010.

[23] C. Bright, V. Ganesh, A. Heinle, I. Kotsireas, S. Nejati, and K. Czarnecki. MathCheck2: A SAT+ CAS verifier for combinatorial conjectures. In V. P. Gerdt, W. Koepf, W. M. Seiler, et al. (Eds.), *Computer Algebra in Scientific Computing - CASC 2016*, LNCS 9890, pp. 117–133, Springer-Verlag, Cham, 2016.

[24] C. Bright, I. Kotsireas, and V. Ganesh. A SAT+CAS method for enumerating Williamson matrices of even order. In S. A. McIlraith and K. Q. Weinberger (Eds.), *AAAI Conference on Artificial Intelligence - AAAI'18*, pp. 6573–6580, AAAI Press, New Orleans, 2018.

[25] B. Buchberger. Bruno Buchberger's PhD thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. *J. Symb. Comput.*, 41, pp. 475–511, 2006.

[26] S. Bulygin and M. Brickenstein. Obtaining and solving systems of equations in key variables only for the small variants of AES. *Math. Comput. Sci.*, 3, pp. 185–200, 2010.

[27] H. K. Büning and T. Lettmann. *Propositional logic: deduction and algorithms.* Cambridge University Press, Cambridge, 1999.

[28] J. Burchard, M. Gay, A. S. Messeng Ekossono, J. Horáček, B. Becker, T. Schubert, M. Kreuzer, and I. Polian. `AutoFault`: Towards automatic construction of algebraic fault attacks. In *Fault Diagnosis and Tolarence in Cryptography - FDTC 2017*, IEEE Computer Society, Taipei, 2017. `https://ieeexplore.ieee.org/document/8167712`.

[29] J. Burchard, A.-S. Messeng Ekosonno, J. Horáček, M. Gay, B. Becker, T. Schubert, M. Kreuzer, and I. Polian. Towards mixed structural-functional models for algebraic fault attacks on ciphers. In *International Verification and Security Workshop - IVSW 2017*. IEEE Computer Society, 2017. `https://ieeexplore.ieee.org/document/8031537`.

[30] D. Choo, M. Soos, K. M. A. Chai, and K. S. Meel. BOSPHORUS: bridging ANF and CNF solvers. arXiv:1812.04580 [cs.LO], 2018. `http://arxiv.org/abs/1812.04580`.

[31] C. Cid, S. Murphy, and M. J. B. Robshaw. Small scale variants of the AES. In H. Gilbert and H. Handschuh (Eds.), *Fast Software Encryption - FSE 2005*, LNCS 3557, pp. 145–162, Springer-Verlag, Berlin, 2005.

[32] M. Clegg, J. Edmonds, and R. Impagliazzo. Using the Groebner basis algorithm to find proofs of unsatisfiability. In G. L. Miller (Ed.), *ACM Symposium on the Theory of Computing - STOC'96*, pp. 174–183, ACM Press, New York, 1996.

[33] N. T. Courtois and J. Patarin. About the XL algorithm over GF(2). In M. Joye (Ed.), *Topics in Cryptology – CT-RSA 2003*, LNCS 2612, pp. 141–157, Springer-Verlag, Berlin, 2003.

[34] N. T. Courtois, P. Sepehrdad, P. Sušil, and S. Vaudenay. ElimLin algorithm revisited. In A. Canteaut (Ed.), *Fast Software Encryption - FSE 2012*, LNCS 7549, pp. 306–325, Springer-Verlag, Berlin, Heidelberg, 2012.

[35] Y. Crama and P. L. Hammer. *Boolean functions: Theory, algorithms, and applications.* Cambridge University Press, Cambridge, 2011.

[36] J. Daemen and V. Rijmen. *The Design of Rijndael.* Springer-Verlag, Berlin, 2002.

[37] A. Dreyer and T. H. Nguyen. Improving Gröbner-based clause learning for SAT solving industrial sized Boolean problems. In *Young Researcher Symposium - YRS*, pp. 72–77, Fraunhofer ITWM, 2013.

[38] D. Eastlake and T. Hansen. US secure hash algorithms (SHA and SHA-based HMAC and HKDF). The Internet Society, RFC 6234, 2011. `https://tools.ietf.org/html/rfc6234`.

[39] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). The Internet Society, RFC 3174, 2001. `https://tools.ietf.org/html/rfc3174`.

[40] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In F. Bacchus and T. Walsh (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2005*, LNCS 3569, pp. 61–75, Springer-Verlag, Berlin, 2005.

[41] N. Een and N. Sörensson. MiniSat: A SAT solver with conflict-clause minimization, 2005. `http://minisat.se`.

[42] J.-C. Faugère. FGb: a library for computing Gröbner bases. In K. Fukuda, J. v. d. Hoeven, M. Joswig, et al. (Eds.), *Mathematical Software - ICMS 2010*, LNCS 6327, pp. 84–87, Springer-Verlag, Berlin, 2010.

[43] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns (Eds.), *Computer Aided Verification - CAV 2007*, LNCS 4590, pp. 519–531, Springer-Verlag, Berlin, 2007.

[44] V. Ganesh, C. W. O'Donnell, M. Soos, S. Devadas, M. C. Rinard, and A. Solar-Lezama. Lynx: A programmatic SAT solver for the RNA-folding problem. In A. Cimatti and R. Sebastiani (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2012*, LNCS 7317, pp. 143–156, Springer-Verlag, Berlin, 2012.

[45] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, 1979.

[46] M. Gay, J. Burchard, J. Horáček, A. S. Messeng Ekossono, T. Schubert, B. Becker, M. Kreuzer, and I. Polian. Small scale AES toolbox: algebraic and propositional formulas, circuit-implementations and fault equations. In *Trustworthy Manufacturing and Utilization of Secure Devices - TRUDEVICE 2016*, Barcelona, 2016. `https://upcommons.upc.edu/handle/2117/99210`.

[47] I. Gaztanaga. The Boost Interprocess Library, 2015. `www.boost.org/doc/libs/1_63_0/doc/html/interprocess.html`.

[48] N. F. Ghalaty, B. Yuce, M. M. I. Taha, and P. Schaumont. Differential fault intensity analysis. In A. Tria and D. Choi (Eds.), *Workshop on Fault Diagnosis and Tolerance in Cryptography - FDTC 2014*, pp. 49–58, IEEE Computer Society, Washington, 2014. `https://ieeexplore.ieee.org/document/6976631`.

[49] J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw. The LED block cipher. In B. Preneel and T. Takagi (Eds.), *Cryptographic Hardware and Embedded Systems - CHES 2011*, LNCS 6917, pp. 326–341, Springer-Verlag, Berlin, 2011.

[50] R. Hao, B. Li, B. Ma, and L. Song. Algebraic fault attack on the SHA-256 compression function. *Int. J. Res. Comput. Sc.*, 4, 2014.

[51] L. Hemme and L. Hoffmann. Differential fault analysis on the SHA-1 compression function. In L. Breveglieri, S. Guilley, et al. (Eds.), *Fault Diagnosis and Tolerance in Cryptography - FDTC 2011*, pp. 54–62, IEEE Computer Society, Washington, 2011.

[52] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In K. Eder, J. Lourenço, and O. Shehory (Eds.), *Hardware and Software: Verification and Testing*, LNCS 7261, pp. 50–65, Springer-Verlag, Berlin, 2012.

[53] J. Horáček, J. Burchard, B. Becker, and M. Kreuzer. Integrating algebraic and SAT solvers. In J. Blömer, I. S. Kotsireas, T. Kutsia, et al. (Eds.), *Mathematical Aspects of Computer and Information Sciences - MACIS 2017*, LNCS 10693, pp. 147–162, Springer-Verlag, Cham, 2017.

[54] J. Horáček and M. Kreuzer. On conversions from CNF to ANF. In M. England and V. Ganesh (Eds.), *2th International Workshop on Satisfiability Checking and Symbolic Computation, SC-square*, Kaiserslautern, 2017. `http://ceur-ws.org/Vol-1974/RP1.pdf`.

[55] J. Horáček and M. Kreuzer. `3BA`: A border bases solver with a SAT extension. In J. H. Davenport, M. Kauers, G. Labahn, and J. Urban (Eds.), *Mathematical Software – ICMS 2018*, LNCS 10931, pp. 209–217, Springer-Verlag, Cham, 2018.

[56] J. Horáček and M. Kreuzer. Refutation of products of linear polynomials. In A. M. Bigatti and M. Brain (Eds.), *3th International Workshop on Satisfiability Checking and Symbolic Computation, SC-square*, Oxford, 2018. `http://ceur-ws.org/Vol-2189/paper9.pdf`.

[57] J. Horáček and M. Kreuzer. On conversions from CNF to ANF. *J. Symb. Comput. (to appear)*, 2019.

[58] J. Horáček, M. Kreuzer, and A. S. Messeng Ekossono. Computing Boolean border bases. In J. H. Davenport, V. Negru, T. Ida, T. Jebelean, et al. (Eds.), *Symbolic and Numeric Algorithms for Scientific Computing - SYNASC 2016*, pp. 465–472, IEEE Computer Society, Timisoara, 2016. `https://ieeexplore.ieee.org/document/7829648`.

[59] J. Horáček, M. Kreuzer, and A. S. Messeng Ekossono. A signature based border basis algorithm. In *Conference on Algebraic Informatics - CAI 2017*, Kalamata, 2017. https://staff.fim.uni-passau.de/kreuzer/papers/SBBBA.pdf.

[60] J. Hsiang. Refutational theorem proving using term-rewriting systems. *Artif. Intell.*, 25, pp. 255–300, 1985.

[61] D. Itsykson and D. Sokolov. Lower bounds for splittings by linear combinations. In E. Csuhaj-Varjú, M. Dietzfelbinger, and Z. Ésik (Eds.), *Mathematical Foundations of Computer Science*, LNCS 8635, pp. 372–383, Springer-Verlag, Berlin, 2014.

[62] K. Jeong, Y. Lee, J. Sung, and S. Hong. Security analysis of HMAC/NMAC by using fault injection. *J. Appl. Math*, 2013, 2013.

[63] P. Jovanovic and M. Kreuzer. Algebraic attacks using SAT-solvers. *Groups Complex. Cryptol.*, 2, pp. 247–259, 2010.

[64] P. Jovanovic, M. Kreuzer, and I. Polian. An algebraic fault attack on the LED block cipher. IACR Cryptology ePrint Archive 2012/400, 2012. https://eprint.iacr.org/2012/400.pdf.

[65] P. Jovanovic, M. Kreuzer, and I. Polian. A fault attack on the LED block cipher. In W. Schindler and S. A. Huss (Eds.), *Constructive Side-Channel Analysis and Secure Design - COSADE 2012*, LNCS 7275, pp. 120–134, Springer-Verlag, Berlin, 2012.

[66] M. G. Karpovsky and Z. Wang. Design of strongly secure communication and computation channels by nonlinear error detecting codes. In *IEEE Transactions on Computers*, 63, pp. 2716–2728. IEEE Computer Society, 2014.

[67] S. Kaspar. Computing border bases without using a term ordering. *Beitr. Algebra Geom.*, 54, 2013.

[68] D. Kaufmann, A. Biere, and M. Kauers. Incremental column-wise verification of arithmetic circuits using computer algebra. *Form. Method Syst. Des.*, 53, 2019.

[69] A. Kehrein and M. Kreuzer. Characterizations of border bases. *J. Pure Appl. Algebra*, 196, pp. 251 – 270, 2005.

[70] A. Kehrein and M. Kreuzer. Computing border bases. *J. Pure Appl. Algebra*, 205, pp. 279–295, 2006.

[71] J. Krajíček. *Proof Complexity.* Cambridge University Press, Cambridge, 2019.

[72] M. Kreuzer and L. Robbiano. *Computational Commutative Algebra 1.* Springer, Heidelberg, 2000.

[73] M. Kreuzer and L. Robbiano. *Computational Commutative Algebra 2.* Springer, Heidelberg, 2005.

[74] O. Kullmann. On a generalization of extended resolution. *Discrete Appl. Math.*, 96-97, pp. 149–176, 1999.

[75] T. Laitinen, T. Junttila, and I. Niemelä. Conflict-driven xor-clause learning (extended version). arXiv:1407.6571 [cs.LO], 2014. `http://arxiv.org/abs/1407.6571`.

[76] M. Lauria, J. Elffers, J. Nordström, and M. Vinyals. CNFgen: A generator of crafted benchmarks. In S. Gaspers and T. Walsh (Eds.), *Theory and Applications of Satisfiability Testing – SAT 2017*, LNCS 10491, pp. 464–473, Springer-Verlag, Cham, 2017.

[77] H. Li and S. Moore. Security evaluation at design time against optical fault injection attacks. In *IEE Proceedings - Information Security*, vol. 153, pp. 3–11. IEEE Computer Society, 2006.

[78] R. Li, C. Li, and C. Gong. Differential fault analysis on SHACAL-1. In L. Breveglieri, I. Koren, D. Naccache, et al. (Eds.), *Fault Diagnosis and Tolerance in Cryptography - FDTC 2009*, pp. 120–126, IEEE Compuer Society, Washington, 2009.

[79] Y. Li, K. Sakiyama, S. Gomisawa, T. Fukunaga, J. Takahashi, and K. Ohta. Fault sensitivity analysis. In S. Mangard and F.-X. Standaert (Eds.), *Cryptographic Hardware and Embedded Systems - CHES 2010*, LNCS 6225, pp. 320–334, Springer-Verlag, Berlin, 2010.

[80] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Learning rate based branching heuristic for SAT solvers. In N. Creignou and D. Le Berre (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2016*, LNCS 9710, pp. 123–140, Cham, 2016. Springer-Verlag.

[81] D. Lokshtanov, R. Paturi, S. Tamaki, R. Williams, and H. Yu. Beating brute force for systems of polynomial equations over finite fields. In R. Krauthgamer (Ed.), *ACM Symposium on Discrete Algorithms - SODA 2016*, pp. 2190–2202, Society for Industrial and Applied Mathematics, Philadelphia, 2016.

[82] F. Marić and P. Janičić. Formal correctness proof for DPLL procedure. *Informatica*, 21, 2010.

[83] B. Mourrain. A new criterion for normal form algorithms. In M. Fossorier, H. Imai, S. Lin, and A. Poli (Eds.), *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, LNCS 1719, pp. 430–442, Springer-Verlag, Berlin, 1999.

[84] S. Nejati, J. Horáček, C. H. Gebotys, and V. Ganesh. Algebraic fault attack on SHA hash functions using programmatic SAT solvers. In J. C. Beck (Ed.), *Principles and Practice of Constraint Programming - CP 2018*, LNCS 11008, pp. 737–754, Springer-Verlag, Cham, 2018.

[85] S. Nejati, J. H. Liang, C. Gebotys, K. Czarnecki, and V. Ganesh. Adaptive restart and CEGAR-based solver for inverting cryptographic hash functions. In A. Paskevich and T. Wies (Eds.), *Verified Software. Theories, Tools, and Experiments - VSTTE 2017*, LNCS 10712, pp. 120–131, Springer-Verlag, Cham, 2017.

[86] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* Wiley-Interscience, New York, 1988.

[87] T. H. Nguyen. *Combinations of Boolean Gröbner bases and SAT solvers.* PhD thesis, TU Kaiserslautern, 2014.

[88] NIST. Secure hash standard (SHS). FIPS 180-4, 2015. `https://csrc.nist.gov/publications/detail/fips/180/4/final`.

[89] J. Nordström. On the interplay between proof complexity and SAT solving. In *ACM SIGLOG News*, 2, pp. 19–44, ACM, New York, 2015.

[90] V. Nossum. SAT-based preimage attacks on SHA-1. Master thesis, University of Oslo, 2012.

[91] V. Nossum. Instance generator for encoding preimage, second-preimage, and collision attacks on SHA-1. In *Proceedings of the SAT competition*, pp. 119–120, 2013.

[92] Opturion. Opturion CPX 1.0.2. `http://cpx.opturion.com/cpx.html`.

[93] J. E. Perry. *Combinatorial criteria for Gröbner bases.* PhD thesis, North Carolina State University, 2005.

[94] I. Polian, M. Gay, T. Paxian, M. Sauer, and B. Becker. Automatic construction of fault attacks on cryptographic hardware implementations. In J. Breier, X. Hou, and S. Bhasin (Eds.), *Automated Methods in Cryptographic Fault Analysis*, pp. 151–170, Springer-Verlag, Cham, 2019.

[95] W. Rankl and W. Effing. *Smart card handbook.* John Wiley & Sons, 2004.

[96] R. Raz and I. Tzameret. Resolution over linear equations and multilinear proofs. *Ann. Pure Appl. Logic*, 155, pp. 194 – 224, 2008.

[97] R. Rivest. The MD5 message-digest algorithm. RFC 1321, 1992.

[98] F. Rossi, P. v. Beek, and T. Walsh. *Handbook of Constraint Programming.* Elsevier Science, New York, 2006.

[99] R. L. Rudell. Multiple-valued logic minimization for PLA synthesis. Defense Technical Information Center, 1986. `https://apps.dtic.mil/docs/citations/ADA606736`.

[100] A. Sabharwal. Algorithmic applications of propositional proof complexity. PhD thesis, University of Washington, 2005.

[101] T. Schubert and S. Reimer. *antom*, 2016. `https://projects.informatik.uni-freiburg.de/projects/antom`.

[102] R. Sebastiani. Lazy satisfiability modulo theories. *J. Satisf. Boolean Model. Comput.*, 3, pp. 141–224, 2007.

[103] I. Semaev. On solving sparse algebraic equations over finite fields. *Des. Codes Cryptogr.*, 49, pp. 47–60, 2008.

[104] R. M. Smullyan. *First-Order Logic*. Springer-Verlag, New York, 1968.

[105] M. Soos. Enhanced gaussian elimination in DPLL-based SAT solvers. In D. L. Berre (Ed.), *Pragmatics of SAT - POS 2010*, EPiC Series in Computing, vol. 8, pp. 2–14, EasyChair, Edinburgh, 2010.

[106] M. Soos. *CryptoMiniSat SAT solver*, 2016. `http://www.msoos.org`.

[107] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full SHA-1. In J. Katz and H. Shacham (Eds.), *Advances in Cryptology – CRYPTO 2017*, LNCS 10401, pp. 570–596, Springer-Verlag, Cham, 2017.

[108] D. R. Stinson. *Cryptography: Theory and Practice*. CRC Press, Boca Raton, 1995.

[109] The ApCoCoA Team. *ApCoCoA: Applied Computations in Commutative Algebra*, 2013. `http://apcocoa.uni-passau.de`.

[110] The Sage Developers. *SageMath: The Sage Mathematics Software System*, 2017. `http://www.sagemath.org`.

[111] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. H. Siekmann and G. Wrightson (Eds.), *Automation of Reasoning: Classical Papers on Computational Logic 1967–1970*, pp. 466–483, Springer-Verlag, Berlin, 1983.

[112] M. Tunstall, D. Mukhopadhyay, and S. Ali. Differential fault analysis of the advanced encryption standard using a single fault. In C. A. Ardagna and J. Zhou (Eds.), *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, LNCS 6633, pp. 224–233, Springer-Verlag, Berlin, 2011.

[113] A. Urquhart. Hard examples for resolution. *J. ACM*, 34, pp. 209–219, 1987.

[114] C. Zengler and W. Küchlin. Extending clause learning of SAT solvers with Boolean Gröbner bases. In V. P. Gerdt, W. Koepf, E. W. Mayr, et al. (Eds.), *Computer Algebra in Scientific Computing - CASC 2010*, LNCS 6244, pp. 293–302, Springer-Verlag, Berlin, 2010.

[115] F. Zhang, S. Guo, X. Zhao, T. Wang, J. Yang, F. Standaert, and D. Gu. A framework for the analysis and evaluation of algebraic fault attacks on lightweight block ciphers. In *IEEE Trans. Information Forensics and Security*, 11, pp. 1039–1054, 2016.

[116] X. Zhao, S. Guo, F. Zhang, Z. Shi, C. Ma, and T. Wang. Improving and evaluating differential fault analysis on LED with algebraic techniques. In W. Fischer and J. Schmidt (Eds.), *Fault Diagnosis and Tolerance in Cryptography - FDTC 2013*, pp. 41–51. IEEE Computer Society, 2013.

Publications

Journal Papers

[1] J. Horáček and M. Kreuzer. On conversions from CNF to ANF. *J. Symb. Comput. (to appear)*, 2019

Remark. This article is the journal version of **[8]**.

Conference Papers

[2] J. Horáček and M. Kreuzer. Refutation of products of linear polynomials. In A. M. Bigatti and M. Brain (Eds.), *3th International Workshop on Satisfiability Checking and Symbolic Computation, SC-square*, Oxford, 2018. `http://ceur-ws.org/Vol-2189/paper9.pdf`

[3] J. Horáček and M. Kreuzer. `3BA`: A border bases solver with a SAT extension. In J. H. Davenport, M. Kauers, G. Labahn, and J. Urban (Eds.), *Mathematical Software – ICMS 2018*, LNCS 10931, pp. 209–217, Springer-Verlag, Cham, 2018

[4] S. Nejati, J. Horáček, C. H. Gebotys, and V. Ganesh. Algebraic fault attack on SHA hash functions using programmatic SAT solvers. In J. C. Beck (Ed.), *Principles and Practice of Constraint Programming - CP 2018*, LNCS 11008, pp. 737–754, Springer-Verlag, Cham, 2018

[5] J. Horáček, J. Burchard, B. Becker, and M. Kreuzer. Integrating algebraic and SAT solvers. In J. Blömer, I. S. Kotsireas, T. Kutsia, et al. (Eds.), *Mathematical Aspects of Computer and Information Sciences - MACIS 2017*, LNCS 10693, pp. 147–162, Springer-Verlag, Cham, 2017

[6] J. Burchard, M. Gay, A. S. Messeng Ekossono, J. Horáček, B. Becker, T. Schubert, M. Kreuzer, and I. Polian. `AutoFault`: Towards automatic construction of algebraic fault attacks. In *Fault Diagnosis and Tolarence in Cryptography - FDTC 2017*, IEEE Computer Society, Taipei, 2017. `https://ieeexplore.ieee.org/document/8167712`

[7] J. Burchard, A.-S. Messeng Ekosonno, J. Horáček, M. Gay, B. Becker, T. Schubert, M. Kreuzer, and I. Polian. Towards mixed structural-functional models for algebraic fault attacks on ciphers. In *International Verification and Security Workshop - IVSW 2017*. IEEE Computer Society, 2017. `https://ieeexplore.ieee.org/document/8031537`

[8] J. Horáček and M. Kreuzer. On conversions from CNF to ANF. In M. England and V. Ganesh (Eds.), *2th International Workshop on Satisfiability Checking and Symbolic Computation, SC-square*, Kaiserslautern, 2017. `http://ceur-ws.org/Vol-1974/RP1.pdf`

[9] J. Horáček, M. Kreuzer, and A. S. Messeng Ekossono. A signature based border basis algorithm. In *Conference on Algebraic Informatics - CAI 2017*, Kalamata, 2017. `https://staff.fim.uni-passau.de/kreuzer/papers/SBBBA.pdf`

[10] M. Gay, J. Burchard, J. Horáček, A. S. Messeng Ekossono, T. Schubert, B. Becker, M. Kreuzer, and I. Polian. Small scale AES toolbox: algebraic and propositional formulas, circuit-implementations and fault equations. In *Trustworthy Manufacturing and Utilization of Secure Devices - TRUDEVICE 2016*, Barcelona, 2016. `https://upcommons.upc.edu/handle/2117/99210`

[11] J. Horáček, M. Kreuzer, and A. S. Messeng Ekossono. Computing Boolean border bases. In J. H. Davenport, V. Negru, T. Ida, T. Jebelean, et al. (Eds.), *Symbolic and Numeric Algorithms for Scientific Computing - SYNASC 2016*, pp. 465–472, IEEE Computer Society, Timisoara, 2016. `https://ieeexplore.ieee.org/document/7829648`