CHALMERS
UNIVERSITY OF TECHNOLOGY

Solving systems of polynomial equations over binary fields:
State-of-the-art algorithms and selected methods.

Master's thesis at the department of Mathematical sciences

Fredrik Meisingseth

Department of Mathematical sciences
Chalmers University of Technology
Gothenburg, Sweden 2022

Solving systems of polynomial equations over binary fields:
State-of-the-art algorithms and selected methods.

Supervisors:
Reinhard Lüftenegger, Graz University of Technology
Christian Rechberger, Graz University of Technology

Examiner:
Anders Södergren

Master's Thesis
Department of Mathematical sciences
Chalmers University of Technology

Solving systems of polynomial equations over binary fields: State-of-the-art algorithms and selected methods.

Fredrik Meisingseth

Abstract

The literature on the topic of methods for solving polynomial systems over $\mathbb{F}_2$ is extensively developed but lacks internal consistency: there are, for example, different choices of formulations of the underlying problem, the notion of complexity and of what assumptions about the input system are warranted. Therefore making comparisons between the different methods can be quite challenging, especially for the novice to the field. In this thesis, an attempt to systemise the knowledge in the field through a survey is made. The survey begins with discussions on, amongst other topics, different formulations of the problem, views on efficiency and reasonable assumptions and then moves on to comparisons between the different methods available. Hopefully these comparisons may serve as a guide for understanding what methods might be most relevant for any specific use case and as a structurising contribution to the literature.

Acknowledgements

First of all, let me express my most genuine gratitude towards my supervisor Reinhard; Our discussions have been highlights of my last five months and have gifted me invaluable insight, not only concerning mathematics but also about many other soul-enhancing matters. I hope to be able to repay at least a small part of that gift of intellectual growth in time. I wish also to thank my other supervisor Christian, for the priviledge of being allowed to write this thesis and for your continued support and encouragement. Further, I thank my examinor Anders for the many points of feedback and counsel. They have made both the thesis a vastly better work and me a better scientist. Finally, I thank my two consultants on foreign language, Charlotte and Carla, the mothers of plants, for consultations on foreign language and asking about the thesis even when you know the answer will start with a deep sigh.

This work was performed at
Graz University of Technology

Keywords: Polynomial system solving, Algebraic cryptanalysis.

Contents

"*No more! Let this be <u>our</u> declaration.*"[1]

[1]- Shoshana Zuboff, *The age of surveillance capitalism*

1 Introduction

1.1 Background

The problem of solving polynomial systems arise in many disciplines of both applied and pure mathematics. This seems natural since from experience we know that polynomial systems are a convenient way to represent different aspects of nature and society. More formally we can build our appreciation of the fundamental role that polynomial systems play in science by that there is a theorem that says that *every function* $f : \mathbb{F}_2^n \to \mathbb{F}_2$ *can be written as a polynomial* (see for example [Zou13]). This means that since all modern computer arithmetics is based on boolean functions, then practically any computation made by a computer can be modelled as a set of polynomial equations. The theorem above is however only one witness to the paramount importance of solving polynomial systems, not only for the scientist but also for society at large, and yet it alone would be enough to motivate the need to explore methods for solving polynomial systems (and especially boolean such) efficiently.

1.1.1 Importance to cryptography

This thesis, whilst at large written agnostically to any specific use-case in the real world, is highly colored (and at times entirely dependent on) that our interest in the methods surveyed primarily stems from *cryptography*. This becomes apparent mostly in how we handle the choice of how to compare different algorithms (see Chapter 4) and in the assumptions we are willing to make about the polynomial system in question (see Chapter 5).

Colloquially, one can say that cryptography is the craft of sending secret messages via a communications channel that is surveilled by an *adversary* (also called *Eve*, an *eavesdropper* or *attacker*). Since the two communicating parties, usually called *Alice* and *Bob*, does not want Eve to know their message, they *encrypt* the message with respect to a *secret key* that they have agreed upon at an earlier timepoint. The idea is now that if the encryption mechanism (or *scheme*) is well designed then it should be very hard (or impossible) for someone without the secret key, like Eve, to learn the original message from the encrypted one.

The question that then is asked is *how hard is it for an attacker to break the scheme, that is, consistently learn original messages without knowing the secret keys?* This question is equivalent to the task of measuring how secure the scheme is and it is answered by a branch of cryptography called *cryptanalysis*. Typically, the security of a cryptographical construction is evaluated by its designers claiming a *security level* of some number b of bits, meaning that for all of the known attacks that the designers have tested on their construction, none of them could break it using less than 2^b operations. Then other cryptanalysts come up with new attacks and if any of them require less than 2^b operations to break the construction then the cryptographical construction is considered broken, and either it or its claimed security level must be adjusted.

One main way of designing attacks is to formulate the calculations in the cryptographical construction as a system of polynomial equations and thus polynomial systems naturally arise, for example, during the analysis of constructions such as hash functions, block ciphers, signature schemes and encryption schemes (like the one described above). As such, and as a consequence of the theorem introduced earlier, also computations that would break these constructions may be formulated as polynomial systems and this means that the security of these constructions inherently and fundamentally relies on that solving such systems is computationally impractical.

This desired hardness does luckily hold. In fact, the problem of solving a general polynomial system

over a given finite field is $\mathcal{NP}$-complete[FY79] and this births the idea that it might be suitable to build certain types of encryption based directly on the hardness of solving such systems. This idea turned out to be very promising, amongst other reasons, due to the current understanding that it seems hard even for quantum computers. The resulting research field of *multivariate public key cryptography (MPKC)* includes schemes such as GeMSS[Nisb], Rainbow [Nisb], HFE[Pat96], UOV[KPG99] and MQDSS [Nisa].

1.2 Purpose and aim

Whilst the study of solving polynomial systems is literally ancient, it is very much still active, see for example the development of the `Crossbred` algorithm [JV18] in 2018 or Dinur's algorithms [Din21][Din] in 2021. Since these new methods seem to constitute a part of the state-of-the-art and are recent enough to, as far as we are aware, not be included in any textbook or considered together at length in another paper or report, the usefulness of us compiling such a report presents itself.

The purpose of this work is therefore to bring structure to the literature on the topic of solving polynomial systems over the binary finite field. This is done with two major objectives in mind:

1. to offer an overview and a map over available methods to the researcher, engineer or student that is about to venture deeper into the literature, and,
2. to offer the cryptanalyst an easy-to-understand guide to the state-of-the-art of available methods.

The methodology chosen to fulfill these aims is to primarily perform a survey over the literature on the topic, summarising and interpreting available methods, and comparing these in a manner such that their respective merits and flaws are showcased. The question that the survey intends to answer is:

Q1: *What is the fastest method for solving a polynomial system over* $\mathbb{F}_2$*?*

1.3 Scope

There are considerable limitations to the scope of this thesis project, primarily stemming from the trade-off between the width and depth of the survey. One quickly recognises that an exhaustive introduction to methods for solving polynomial systems for many reasons is out of reach. Even an exhaustive introduction to the role of polynomial systems in cryptography must be seen as grasping for too much, although much closer to what could be realistic. Then finally, one lands at the inquiry into some of the methods for solving polynomial systems that are most significant from the point of view of cryptanalysis, namely those operating over $\mathbb{F}_2$. This inquiry, whilst evidently possible to fruitfully carry out as a master's thesis, is still not exhaustive. Most notably perhaps, we opted to not implement the methods included in the survey in order to benchmark them against each other in practice. Whilst this would possibly have been a valuable addition to the understanding of how the methods relate to each other, it would shift the focus from *why* the methods exhibit different properties regarding, for example, their performance and assumptions towards more confirming *that* they indeed are.

1.4 Contributions

The main scientific contribution of this work is, as it typically is with surveys, that it gives an overview of the topic in question and offers a guide to the literature. This is done partly through summaries

of papers and partly through interpretations of them. Whilst the descriptions, re-structurings and formulations are mostly new in this work, one must note that during the survey more or less no new ideas are introduced. Rather are the analyses that we make to be seen as interpretations of previously made analyses on specific parts of the topic. The conclusions drawn are suitably seen as us giving our take on arguments given by previous works, rather than us offering new arguments.

Additionally to the literature survey, we also improve an implementation of one of the algorithms surveyed. This is however to be seen as an extension of the survey, in that its primary aim is to bring further understanding about the algorithm in question rather than being optimised enough for practical use. The contribution from this update of an existing implementation should in any case be seen as minor, partly because the work we have contributed to the code is small in comparison to the authors of the previous version and partly because there are still much development needed before the implementation is ready for thorough testing or external use.

1.5 Structure

The rest of this report is structured as follows: Chapter 2 introduces some preliminaries preparing the reader for the upcoming chapters and chapter 3 presents the formalities of the problem of solving polynomial systems. Chapters 4 and 5 consist of remarks; first on how one should navigate the stormy waters of comparing methods that measure slightly different things and then secondly about the choices and assumptions we make during the rest of the report. Chapters 6 to 10 summarises the four families of methods that play a part in the state-of-the-art. Chapter 11 describes the improvements we make to the implementation of an algorithm and chapter 12 constitute a performance comparison between the methods considered in the earlier chapters, including translations of results from one model of computation to another. The final chapter consist of conclusions, discussions of these and outlooks.

2 Preliminaries

2.1 Notation

We write vectors in bold, for example $\mathbf{x} := (x_1, ..., x_n)$. Concatenation of vectors will be done with abuse of notation with the hope that the context in itself is enough, for example, we will denote the concatenation of two vectors $\mathbf{y}, \mathbf{z}$ by $(\mathbf{y}, \mathbf{z})$. $HW(\mathbf{x})$ is the hamming weight of some variable $\mathbf{x}$ in a binary representation and we denote probability by $\mathbb{P}$. In pseudocode, let "$\leftarrow$" denote assignment to a variable. Let $\binom{a}{\downarrow b} := \sum_{i=0}^{b} \binom{a}{i}$.

2.2 Algebra

The formulations in this subchapter are primarily reformulations of the definitions given in the book Algebra 1 [Bou89] by the mathematical collective Nicolas Bourbaki.

Intuitively, one can with merit think of the *finite field*, also called *Galois field*, $\mathbb{F}_q$ of q elements, $q := p^n$ for p prime and $n \in \mathbb{N}$, as simply what happens if one operates on the whole numbers $\mathbb{Z}$ and then takes $mod(q)$ on all calculations. In order to formally introduce the notion of a finite field, we must first consider some underlying definitions.

Definition 2.1 (Group, Abelian group)**.**
A group $(G, +)$ is a set G with a mapping $+ : G \times G \to G$, usually called addition, that for all elements $a, b, c \in G$, satisfies the following properties:

1. *Addition is associative:* $(a + b) + c = a + (b + c)$.
2. *There is an identity element with respect to addition, called* 0*, in G such that:* $a + 0 = 0 + a = a$.
3. *Each element $a \in G$ has an inverse with respect to addition, denoted $-a$, such that:* $a + -a = -a + a = 0$.

The group is called abelian if addition also is commutative, that is if, for all $a, b \in G$: $a + b = b + a$.

Definition 2.2 (Ring)**.**
A ring $(R, +, \cdot)$ is a set R with two operations $+$ and $\cdot$, called addition and multiplication, that for all elements $a, b, c \in R$ satisfies the following properties:

1. *$(R, +)$ is an abelian group.*
2. *Multiplication is associative:* $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
3. *There is an identity element, called* 1*, in R with respect to multiplication, meaning:* $a \cdot 1 = 1 \cdot a = a$.
4. *Multiplication is distributive with respect to addition:* $a \cdot (b + c) = a \cdot b + a \cdot c$ *and similarily* $(a + b) \cdot c = a \cdot c + b \cdot c$.

The ring is called commutative if it is also commutative with respect to multiplication.

Definition 2.3 (Field, $\mathbb{F}$)**.**
A commutative ring $\mathbb{F} := (F, +, \cdot)$ is called a field if F does not consist of only 0 and if every non-zero element of $\mathbb{F}$ is invertible with respect to multiplication.

A field is called a finite field if it contains only a finite number of elements.

A helpful intuition behind these algebraic constructs is to think that a group is a set together with addition and subtraction, whilst a ring also has multiplication and finally a field also has division.

Definition 2.4 (Homomorphism, Bijection, Isomorphism)**.**
Let A and B be two rings. A (ring) homomorphism of A to B is any mapping $f : A \to B$ such that:

$$f(x + y) = f(x) + f(y),$$
$$f(x \cdot y) = f(x) \cdot f(y),$$
$$f(1) = 1.$$

The homomorphism f is called bijective iff each element in B is mapped to by exactly one element in A. The homomorphism f is an (ring) isomorphism iff it is a bijective homomorphism.

Theorem 2.1 (Galois field, Theorem 7.1.3 in [Mor03])**.**
For each prime number p, and for each $n \in \mathbb{N}$, there is a unique (up to an isomorphism) finite field $\mathbb{F}_q$ of cardinality $q = p^n$, and it is called the Galois field with q elements.

The important take away from this theorem is that if we regard two fields as equal, or the same, if they are isomorphic to each other then there is a unique finite field of each size. For the continuation of this report, we will refer to the Galois field of cardinality q as the finite field of size q, and denote it $\mathbb{F}_q$. Additionally one should note that we will talk about elements in the set of the field $\mathbb{F}$ simply as elements in $\mathbb{F}$.

Definition 2.5 (Polynomial ring)**.**
The set of all polynomials in several variables with coefficients in a field $\mathbb{F}$ forms a commutative ring, called a polynomial ring over $\mathbb{F}$ and is denoted $\mathbb{F}[\mathbf{x}]$, where $\mathbf{x}$ is a vector of formal variables in the elements of $\mathbb{F}$.

Definition 2.6 (Ideal)**.**
Let R be a commutative ring. A subset $\mathcal{I}$ of R is called an ideal if it is a subgroup of the additive group of R and the relations $a \in R$ and $f \in \mathcal{I}$ together imply $af \in \mathcal{I}$.

For the case of an ideal generated by a finite set of polynomials $E := \{P_j(\mathbf{x})\}_{j=1}^m$, one can note that this ideal may similarily be defined as

$$\mathcal{I} := \langle P_1, ..., P_m \rangle := \{\sum_{j=1}^{m} h_j P_j | \forall (h_1, ..., h_m) \in \mathbb{F}[\mathbf{x}]^m\}. \tag{1}$$

For the case of polynomial systems, one can note that the idea of an ideal of the polynomials is analog to the idea of the set of all linear combinations of the polynomials of the system.

2.3 Boolean algebra

The notations of the previous subchapter lets us express that a polynomial system E of m polynomials in n boolean variables $\mathbf{x} := (x_1, ..., x_n)$ means that $\mathbf{x} \in \mathbb{F}_2^n$ and $E \in \mathbb{F}_2[\mathbf{x}]^m$, i.e. it is a subset of the polynomial ring $\mathbb{F}_2[\mathbf{x}]$.

One special property of $\mathbb{F}_2$ is the fact that $x^2 = x$ and this equation is sometimes called the *field equation* of $\mathbb{F}_2$. Due to the field equations for $\mathbb{F}_2^n$ we may regard polynomials that can be reduced to each other (via modulus of the field equations) as equivalent, and for example say $x_1 \cdot x_2^5 = x_1 \cdot x_2$, since they will yield the same output for any possible input and thus they compute the same function. As a consequence of this view of equivalence, some authors prefer to formally let the polynomial system E belong to a quotient ring of $\mathbb{F}_2[\mathbf{x}]^m$, namely $\mathbb{F}_2[\mathbf{x}]^m/[x_1^2 - x_1, ..., x_n^2 - x_n]$. Reasons for including the field equations can for example be that the algorithm in question thrives from having a larger number of equations to satisfy. However since all evaluations of a polynomial remains unchanged by the modulus of the field equations, one need not be worried by that some algorithms assume the input polynomials to be over the quotient ring and some do not.

The above mentioned property of $\mathbb{F}_2$ lets us note that the polynomials of E may be written in a particularily convenient way, let once again

$$E := \{P_j(\mathbf{x})\}_{j=1}^m \tag{2}$$

be a system of m polynomials of n variables over the field $\mathbb{F}_2$. Then each polynomial may be stated as a sum of monomials

$$P_j(\mathbf{x}) := \sum_{\mathbf{u} \in \mathbb{F}_2^n} \alpha_{j\mathbf{u}} \prod_{i=1}^n x_i^{u_i}, \tag{3}$$

$\alpha_{j\mathbf{u}} \in \mathbb{F}_2$. We will for this entire thesis use this specific way of notating the input system to be solved, and will reserve the use of all included variable namings for this purpose. We will at times write $\mathbf{x}^\mathbf{u}$ instead of $\prod_{i=1}^n x_i^{u_i}$. We define the *degree of a monomial* as $deg(\mathbf{x}^\mathbf{u}) := \sum_{i=1}^n u_i$, which also is the hamming weight of $\mathbf{u}$, and the *degree of a polynomial* is defined as the maximum degree of its constitutionary monomials.

Another noteworthy property of $\mathbb{F}_2$ is that both elements are their own additive inverse, namely that $x + x = 0$ and thus $x = -x$.

2.4 The Möbius transform

This subchapter is based on Chapter 9.2 in [Jou09] and the explaination of the Möbius transform in [Din21].

The form of writing a polynomial according to (3) is in fact unique, if we say that the polynomial is defined by its truthtable, and this form is called the *algebraic normal form (ANF)* of P_j. Since the truthtable then defines the coefficients $\alpha_{j\mathbf{u}}$ we may define the function that maps $\mathbf{u}$ to $\alpha_{j\mathbf{u}}$. For an arbitrary binary polynomial P written as $P(\mathbf{x}) = \sum_{\mathbf{u} \in \mathbb{F}^n} \alpha_\mathbf{u} \prod_{i=1}^n x_i^{u_i}$, we say $\alpha_\mathbf{u} := g(\mathbf{u})$ for a function g that we will call *the Möbius transform* of P. For binary functions over $\mathbb{F}_2^n$, the Möbius transform is its own inverse and is thus in this case also at times referred to by the name of its inverse, namely *the zeta transform*.

The Möbius transform can be calculated by noting that a polynomial P can always be written as a sum of two other polynomials $P^{(0)}$, $P^{(1)}$ by

$$P(\mathbf{x}) = P^{(0)}(x_1, ..., x_{n-1}) + x_n \cdot P^{(1)}(x_1, ..., x_{n-1}). \tag{4}$$

The respective Möbius transforms of P, $P^{(0)}$ and $P^{(1)}$ are then related as

$$g(x_1, ..., x_{n-1}, 0) = g^{(0)}(x_1, ..., x_{n-1}), \tag{5}$$

$$g(x_1, ..., x_{n-1}, 1) = g^{(1)}(x_1, ..., x_{n-1}), \tag{6}$$

and since further P, $P^{(0)}$ and $P^{(1)}$ satisfies

$$P^{(0)}(x_1, ..., x_{n-1}) = P(x_1, ..., x_{n-1}, 0), \tag{7}$$
$$P^{(1)}(x_1, ..., x_{n-1}) = P(x_1, ..., x_{n-1}, 0) + P(x_1, ..., x_{n-1}, 1), \tag{8}$$

one can recursively calculate the Möbius transform from the truth table of P. This may be done in $n2^n$ field operations.

Since the Möbius transform is its own inverse it facilitates both the translation from the truthtable to the ANF, i.e. *interpolation*, and the other way around, i.e. *evaluation*. Interestingly, the Möbius transform does not require the entire truthtable to correctly interpolate P. Important for later chapters is the result (see [Din] or [KKW16]) that if P has degree at most d, then it can be interpolated solely from the truthtable of the inputs in $W_d^n := \{\mathbf{x} \in \mathbb{F}_2^n | HW(\mathbf{x}) \leq d\}$. This means that the ANF of a polynomial can be interpolated using at most $n\binom{n}{\downarrow d}$ field operations, potentially much less than the $n2^n$ required if one uses the entire truthtable of the polynomial.

2.5 Valiant-Vazirani affine hashing

This subchapter is based on Chapter 2.5 in [Bar+21] and Chapter 2.5 in [BKW19].

Let S be the set of solutions to E and say that one wants to *isolate* a solution, which intuitively means to translate E into a slightly different system E' such that one knows that if one finds a solution to that system then it is both the only solution to E' and a solution to E. Such a system E' can be constructed using *Valiant-Vazirani affine hashing*[VV86], which roughly goes like this: Say c is the unique integer such that $|S| \in [2^c, 2^{c+1})$. Generate $c + 2$ linear polynomials $p_1, ..., p_{c+2}$ by drawing $a_{ij}, b_j \in \mathbb{F}_2$ independently, uniformly random and defining p_j by

$$p_j(\mathbf{x}) := \sum_{i=1}^{n} a_{ij} x_i - b_j.$$

These polynomials are then appended to E, thus forming $E' := \{P_1, ..., P_m, p_1, ..., p_{c+2}\}$. The point of this construct is that the probability of a solution $\mathbf{x}^* \in S$ being the only one that also is a solution of E' is at least $\frac{1}{2^{c+3}}$. Thus by repeatedly creating new E' one can isolate a solution with arbitrarily high probability. Critically, the number of equations that needs to be added is upper bounded by $log_2(n)$, meaning that the comparative size of the new system after isolating solutions is not much larger than the original one. For more details on this construction, see [BKW19] or [VV86].

2.6 Gröbner bases

This subchapter is based on Chapter 11 in the 2009 book[Jou09] by Joux and Chapter 2 in the 2007 book[CLO07] by Cox, Little and O'Shea.

The aim that leads to the construction of *Gröbner bases* [Buc06] is that of, rather than solving the given system E directly, translating E into another system E' that is somehow easier to solve but has the same roots. A Gröbner basis is precisely such a system E' and is found by carefully building a set of polynomials that generate the same ideal as E.

In order to formally give the definition of a Gröbner basis, we first present some supporting constructions. Note that a monomial in a polynomial ring $\mathbb{F}[\mathbf{x}]$ is of the form $x_1^{u_1} \cdot ... \cdot x_n^{u_n}$, with $\mathbf{u} \in \mathbb{N}^n$. Adopting vector notation, we write a monomial as $\mathbf{x}^{\mathbf{u}}$.

Definition 2.7 (Monomial ordering)**.**
We call a total ordering relation $\succeq$ on the set of monomials of $\mathbb{F}[\mathbf{x}]$ a monomial ordering if it satisfies the following properties:

- ***Compatibility:*** *For any triple $(\mathbf{x}^\alpha, \mathbf{x}^\beta, \mathbf{x}^\gamma)$ of monomials:*

$$\mathbf{x}^\alpha \succeq \mathbf{x}^\beta \implies \mathbf{x}^\alpha \mathbf{x}^\gamma \succeq \mathbf{x}^\beta \mathbf{x}^\gamma$$

  .

- ***Well-ordering:*** *Any set S of monomials contains a least (smallest) element with respect to $\succeq$, and that element is denoted by $min_\succeq(S)$.*

Definition 2.8 (Lexicographic ordering, lex)**.**
The lexicographical ordering lex, denoted by that $\succeq_{lex}$, is defined by that $\mathbf{x}^\alpha \succ_{lex} \mathbf{x}^\beta$ iff $\exists i_0 \in \{1, ..., n\}$ such that: $\forall i < i_0 : \alpha_i = \beta_i$ and $\alpha_{i_0} > \beta_{i_0}$.

Definition 2.9 (Reversed lexicographic ordering, revlex)**.**
The reverse lexicographical ordering revlex, denoted by $\succeq_{revlex}$, is defined by that $\mathbf{x}^\alpha \succ_{revlex} \mathbf{x}^\beta$ iff $\exists i_0 \in \{1, ..., n\}$ such that: $\forall i < i_0 : \alpha_i = \beta_i$ and $\alpha_{i_0} < \beta_{i_0}$.

Definition 2.10 (Graded reversed lexicographic ordering, grevlex)**.**
The graded reverse lexicographical ordering grevlex, denoted by $\succeq_{grevlex}$, is defined by that $\mathbf{x}^\alpha \succ_{grevlex} \mathbf{x}^\beta$ iff either
$deg(\mathbf{x}^\alpha) > deg(\mathbf{x}^\beta)$
or
$deg(\mathbf{x}^\alpha) = deg(\mathbf{x}^\beta)$ and $\mathbf{x}^\alpha \succ_{revlex} \mathbf{x}^\beta$.

Let $LM(P_j)$ denote the *leading monomial* of P_j, i.e. the largest monomial in P_j with respect to some monomial ordering $\succeq$, and $LM(\mathcal{I})$ be the ideal generated by the leading monomials of all polynomials in some set with ideal $\mathcal{I}$. Similarily let $LC(P_j)$ denote the *leading coefficient* of P_j, i.e. the coefficient of the leading monomial, and let $LT(P_j) := LC(P_j)LM(P_j)$ be called the *leading term* of P_j. We may now give the definition of a Gröbner basis.

Definition 2.11 (Gröbner basis)**.**
A set of polynomials $\{g_1, ..., g_k\}$ is a Gröbner basis for an ideal $\mathcal{I}$ iff

(i) $\mathcal{I} = \langle g_1, ..., g_k \rangle$,

(ii) $LM(\mathcal{I}) = \langle LM(g_1), ..., LM(g_k) \rangle$.

2.6.1 Calculating the roots of E from a Gröbner basis

Note that the translation of a Gröbner basis to a root of E is highly dependent on the used monomial ordering, more specifically is the lex ordering needed [Jou09][Fau+93].

First of all, we can note that if the Gröbner basis includes 1, then there are no roots of E since the non-zero constant polynomial never can evaluate to 0. Secondly, if there is a finite positive number of roots then the Gröbner basis will be of the form such that for each x_i, there is a polynomial in the Gröbner basis whose leading monomial includes a power of x_i, and also that polynomial has no terms including any x_j with $x_j \succeq x_i$. Thus one can obtain a root of E by first solving the polynomial in the Gröbner basis that only involves x_n, then the one only including $\{x_n, x_{n-1}\}$ and so on.

2.6.2 Buchberger's algorithm

In his 1965 thesis [Buc06], Buchberger does not only introduce the idea of Gröbner bases, he also gives a first algorithm for finding one. This algorithm, creatively remembered as *Buchberger's algorithm*, is a foundation that many later algorithms build upon and is thus of prime scholarly interest. Below in algorithm 1 a simplified formulation of the algorithm is presented, but first some supporting definitions.

Definition 2.12 (S-polynomial, reformulation of Definition 4, Chapter 2.6 in [CLO07])**.**
Let $f, g \in \mathbb{F}[\mathbf{x}]$ be non-zero. Then there exists a third polynomial $S(f,g)$, called their S-polynomial that is defined as such:

$$S(f,g) = \frac{lcm(LM(f), LM(g))}{LT(f)} f - \frac{lcm(LM(f), LM(g))}{LT(g)} g,$$

where $lcm(LM(f), LM(g)) := \mathbf{x}^{\gamma}$ and $\gamma_i := max(\alpha_i, \beta_i)$ where α and β are the coefficient vectors of $LM(f)$ and $LM(g)$ respectively.

A useful intuition regarding the point of constructing S-polynomials is that the two constitutionary terms share the same leading term, and thus the leading term is cancelled.

Definition 2.13 (Remainder after division of ordered set of polynomials, Reformulation of Definition 3, Chapter 2.6 and Theorem 3, Chapter 2.3 in [CLO07].)**.**
Let $G = \{g_1, ..., g_k\}$ be an ordered set of polynomials in $\mathbb{F}[\mathbf{x}]$. Then every $f \in \mathbb{F}[\mathbf{x}]$ can be written as

$$f = q_1 g_1 + ... + q_k g_k + r, \tag{9}$$

where $q_1, ..., q_k, r \in \mathbb{F}[\mathbf{x}]$ and either $r = 0$ or r is a linear combination, with coefficients in $\mathbb{F}$, of monomials, none of which is divisible by any of $LT(g_1), ..., LT(g_k)$. r is called the remainder on division of f by the ordered set G. The remainder is also denoted $\overline{f}^G$.

In the upcoming chapters we will also at times call $\overline{f}^G$ *f reduced with respect to G*. Remainders of S-polynomials play a key role in Buchberger's algorithm and lead nearly directly to it through the next result.

Theorem 2.2 (Buchberger's criterion, Theorem 6, Chapter 2.6 in [CLO07])**.**
Let $\mathcal{I}$ be a polynomial ideal. Then a basis $G = \{g_1, ..., g_k\}$ of $\mathcal{I}$ is a Gröbner basis of $\mathcal{I}$ iff $\forall i \neq j$,

$$\overline{S(g_i, g_j)}^G = 0. \tag{10}$$

Hence the overarching idea of Buchberger's algorithm appears, namely to start with the input system and then iteratively reducing pairs of the set with respect to the set and appending their remainder to the set, if non-zero, and then repeating until all remainders are zero. Pseudocode for Buchberger's algorithm follows below in Algorithm 1 , which is a reformulation of the one given in [CLO07].

Algorithm 1 Buchberger's algorithm

Require: A list G of polynomials.
$G' \leftarrow \emptyset$
while $G' \neq G$ **do**
  $G' \leftarrow G$
  for each pair $g_i, g_j \in G$, $g_i \neq g_j$ **do**
    $r \leftarrow \overline{S(g_i, g_j)}^G$
    if $r \neq 0$ **then**
      $G \leftarrow G \cup \{r\}$
    end if
  end for
end while
Output: Gröbner basis G

2.7 Macaulay matrices

The *Macaulay matrix*[Mac16] of a polynomial system, or more correctly of the *ideal generated by a polynomial system*, is a representation of said system in matrix form. Due to Macaulay's complicated frasing and framing we base the following explaination on the one given in Chapter 11.5 in [Jou09]. The construction is as such:

Suppose we have a system E of m polynomials $\{P_1, ..., P_m\}$ of degree d or less in n variables $(x_1, ..., x_n)$. Fix some degree $D \geq d$, then the Macaulay matrix of degree D, $M_{\leq D}$, for E is the matrix whose columns are labeled by all monomials in $\mathbf{x}$ of degree no larger than D and whose rows are labeled by single multiples of a polynomial and a monomial such that the product has degree at most D. There are $k := \binom{n+D}{D}$ such monomials. If we call the set of the monomials $T_D := \{t_1, ..., t_k\}$ then we can note that the columns of $M_{\leq D}$ are indexed by $t_1, ..., t_k$ and if $deg(t_i) + deg(P_j) \leq D$, then there is a row of $M_{\leq D}$ labeled $t_i P_j$. The elements in the matrix are then the corresponding coefficients of the products $t_i P_j$ and if we let α_{jl} denote the coefficient of t_l in P_j, then the matrix may be written as

$$M_{\leq D} := \begin{array}{c} \\ t_i P_j \\ \vdots \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & \cdots & t_k \end{array} \\ \begin{pmatrix} \alpha_{j1} & \alpha_{j2} & \cdots & \alpha_{jk} \\ & & \ddots & \end{pmatrix} \end{array}.$$

The construction is perhaps even clearer in a concrete example:

Say $E = \{P_1 := 2x + y, P_2 := y^2\}$ and $\mathbf{x} := (x, y)$. Fix $D = 2$. Say $T := \{x^2, xy, y^2, x, y, 1\}$. Then we get

$$M_{\leq 2} = \begin{array}{c} \\ 1P_1 \\ xP_1 \\ yP_1 \\ 1P_2 \end{array} \begin{array}{c} \begin{array}{cccccc} x^2 & xy & y^2 & x & y & 1 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 2 & 1 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

2.8 Computational complexity theory

This subchapter is primarily based on the 2009 book "Computational complexity: a modern approach"[AB09] by Arora and Barak, the 1974 book "The design and analysis of computer algorithms"[AHU74] by Aho, Hopcraft and Ullman and on the 2022 paper [Bou22] by Bouillaguet.

Computational complexity theory is the field of mathematics stringently defining the hardness of problems and the performance of algorithms. Both of these aspects are of significance for us in our inquiry following in the upcoming chapters; Firstly, we describe that the problem of solving a polynomial system of degree 2 or more belongs to the complexity class $\mathcal{NP}$-*complete* and secondly we will compare different measures for assessing the performance of algorithms for solving the problem.

The hardness classification of problems in practice primarily takes the shape of deciding whether the problem belongs to the complexity class $\mathcal{P}$, the class of problems that can be solved in a polynomial number of steps by a *deterministic Turing machine* (`TM`), or that of $\mathcal{NP}$, the class of problems that can be solved in a polynomial number of steps by a *non-deterministic Turing machine* (`NTM`), or neither. Equivalently $\mathcal{NP}$ can be defined by as the set of problems for which a proposed solution can be *verified* by a `TM` in a polynomial number of steps. It is therefore obvious that $\mathcal{P}$ is a subset of $\mathcal{NP}$, but the question whether it is a proper subset or if $\mathcal{P} = \mathcal{NP}$ is an incredibly important open problem in theoretical computer science and goes under the amply chosen name of *the $\mathcal{P}$ versus $\mathcal{NP}$ problem*. The importance of the problem can for example be seen in that it is listed as one of the millenium problems by the *Clay mathematics institute*, a list of problems whose solution is rewarded by a prize of one million dollars. It is commonly believed that $\mathcal{P}$ is <u>not</u> equal to $\mathcal{NP}$, and this belief we will call *the P-vs-$\mathcal{NP}$ conjecture* [AB09].

The hardest problems in $\mathcal{NP}$, in the sense of that a solution to any of these problem leads efficiently (in polynomial time) to a solution to any other problem in $\mathcal{NP}$, are called $\mathcal{NP}$-*complete* and problems that are shown to be at least this hard are called $\mathcal{NP}$-*hard*. The interest in these complexity classes from the point of view of cryptography stems from the fact that $\mathcal{NP}$-complete problems, under the $\mathcal{P}$-vs-$\mathcal{NP}$ conjecture, are unpractical to solve whilst still allowing for that confirming a supposed solution can be done efficiently. Thus such problems may conveniently serve as the basis for cryptographical schemes, primarily asymmetric ones. Amongst other constructions is the $\mathcal{P}$-vs-$\mathcal{NP}$ conjecture closely linked to the conjectured existence of one-way functions.

From the discussion above, it becomes clear that the hardness of a problem is entirely dependent on the choice of the underlying computational model, for example does it matter fundamentally if one evaluates the difficulty of a problem with respect to a `TM` or to a `NTM`. Similarily critical is the chosen model of computation when one evaluates the performance of an algorithm. Therefore the next subchapter will shortly present some models of computation that will be used in the upcoming discussions.

First however, let us define what we mean by the complexity of an algorithm:

Definition 2.14 (Time complexity of an algorithm (informal) , summary of Chapter 1.1 in [AHU74])**.** *Let N be some fixed positive integer denoting the size of the input to some algorithm $\mathcal{A}$. Let $\mathcal{M}$ be some model of computation. The <u>time complexity</u> of $\mathcal{A}$ as a function of N under the model $\mathcal{M}$ is then the time that is needed for a machine of type $\mathcal{M}$ to perform the instructions of $\mathcal{A}$ for an input of size N.*

Analogously, one may define the *space complexity*. Of special importance is the behaviour that the complexity, as a function of N, exhibits as N grows to infinity. This is called the *asymptotic complexity*,

but we will with some abuse of notation in the future refer to this simply as the complexity since it gives a more descriptive view of the performance of the algorithm in the sense that it does not rely on some specific choice of N.

Whilst the above definition well conveys the intuition we have about the complexity of an algorithm, one quickly notices the ambiguity of the formulation. For example, we have not given definitions of what is meant by critical words such as "algorithm", "machine" and "time". Diving into a fully stringent definition of these notions, and the idea of complexity, is a highly rewarding scholarly quest and offers not only a deep theoretical understanding of contemporary computer science but also of its historical origins. It suffices however for our purposes to keep the intuitive idea that an algorithm is a mathematical "recipe" for calculating a function, and that this algorithm is then translated to a *program*, a list of *instructions*, in the model of computation and that the model of computation stringently specifies what time is associated with each instruction. Similarily, the notion of the size of the input to the program can suitably be kept conveniently vague.

Clear from the definition above, however, is the critical role that specifying a model of computation plays in the notion of the complexity of an algorithm.

One should also note the vagueness about what specific input is meant when talking about the complexity of an algorithm, since it is obvious that different problem instances of the same size might incur different time requirements. Two useful alternatives appear here; the idea of *worst-case complexity*, meaning that *any* input of the given size may be solved in the presented time, and the idea of *expected complexity* (also called *average-case complexity*), meaning that one treats the needed time as a stochastic variable and considers its expected value under some assumptions of the input space. Both of these have their merits and are present in the discussions in the later chapters of this report. For simplicity though, when not specified, we do mean the worst-case complexity.

2.9 Models of computation

Just as the choice of computational model is highly debatable and ultimately up to the user, the presentations and specifications of the same models are highly variable. The following presentations are to a large degree based on [AHU74], but the reader should bear in mind that the formulations chosen are very much not the only ones in contemporary use. Alternative definitions, and much useful insights, are to be found in the freely available 1997 book by John Savage [Sav97].

Let us first note that for our purposes a model of computation is simply an unambigous tool for going from an algorithm (a list of some instructions) to a complexity. Therefore it suffices to define which instructions are available, how one may combine these and how they are to be interpreted in terms of time costs. Whilst this is the core, much complications are at times added in order to either remove possibilities for ambiguity or to ensure that the rules for how the instructions may be used well represents some idea of how a real computer works.

2.9.1 Turing machine (TM) model

As outlined in the previous chapter, much of complexity theory relies on the mathematical construct of a machine called a *Turing machine (TM)* that was deviced by Alan Turing in 1937 [Tur37]. The paramount usefulness of this construct can not be overstated and due to its continued importance, not the least as a defining part of commonly used complexity classes, we will now shortly introduce it even though it will not be part of the discussions of the latter parts of this work. For a more complete description of the turing machine, see [Tur37], [Sav97] or [AHU74].

Definition 2.15 (Turing machine (TM) (informal), reformulation of the definition in Chapter 1.6 in [AHU74])**.**
A Turing machine (TM) consists of an infinite tape and a finite state program. The tape is divided into cells, each of which holds one out of a finite set of tape symbols. The program controls a tape head, which may read or write a cell of the tape. The program is in a state, of which there are finitely many, and this decides the operation to be taken by the machine.

In one step of the execution of the program, the TM may perform any of the operations:

- *Change the state of the program.*
- *Read/write a tape symbol over the symbol at the tape cell under the tape head.*
- *Move the tape head one place to the left or right.*

The state of the program decides which operations to perform, and the transition between states is ruled by a transition function. Execution of the program continues until a specific final state has been reached.

A specific TM is thus fully defined by the set of states, the set of tape symbols, the symbols at the tape at the start of the execution, the initial state, the final state and the transition function. A NTM is defined analogously but with the difference that the transition function is stochastic rather than deterministic. An illustration of a TM can be seen in Figure 1 below.

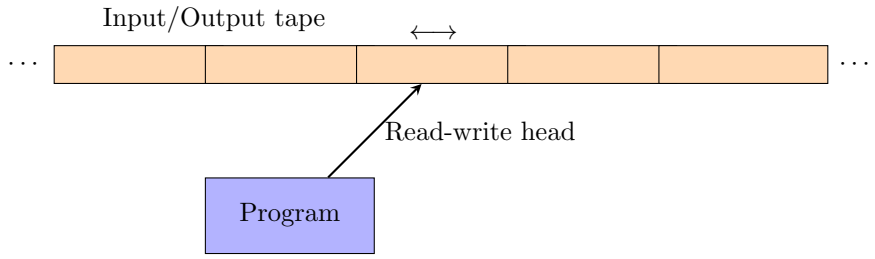

Figure 1: An illustration of a TM.

Whilst the TM has proven useful for proving properties of problems, and to a lesser degree algorithms, its unwieldy formalism and low degree of correspondence to the architectures of modern computers led to other models being introduced, such as the RAM model.

2.9.2 Random Access Machine (RAM) model

The *Random Access Machine (RAM)* model, originally proposed by Cook and Reckhow in 1973[CR73], may be seen as a mathematical representation of the von Neumann computer architecture and is thus a convenient choice for the computer scientist. The RAM model describes a computer with one read-only input tape, one write-only output tape, a memory consisting of an unlimited amount of registers each holding an integer of unlimited size, all of which is controlled by a program. There are other definitions of a RAM, for example detailing the way in which the program exerts control of the memory, but we have

opted for the formulation in [AHU74], which closely resembles the one by Cook and Reckhow. Whilst later reformulations perhaps have a closer recemblance to modern data architectures, the difference causes more or less no change to complexity results. A visualisation of a RAM can be seen in Figure 2.

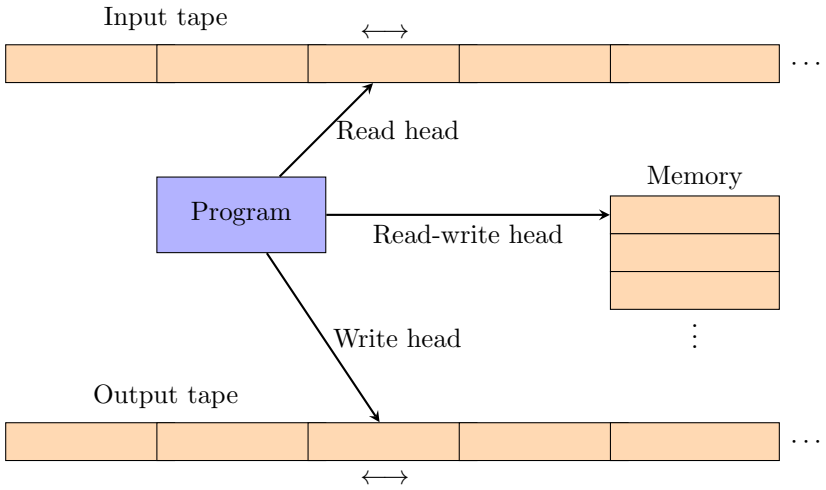

Figure 2: An illustration of a RAM.

The program is a list of *instructions* belonging to an *instruction set*, for example

Instruction name	**Description**
READ	Read the square of the input tape at the current place of the read-pointer and increase the pointer by 1. Store the read value into register 0.
WRITE	Write the value of register 0 onto the output tape at the position of the write-pointer and increase the pointer by 1.
LOAD	Load the value of some specified register to register 0.
STORE	Store the value of register 0 to some other specified register.
ADD	Add the value of some specified register to the value of register 0 and store the result in register 0.
SUB	Analogous to ADD but with subtraction.
MUL	Analogous to ADD but with multiplication.
DIV	Analogous to ADD but with division (result rounded down).
JUMP	The location counter is moved to a specified place.
JGTZ	If the value of register 0 is greater than 0, then the location counter is set to a specified location and other wise it is increased by 1.
JZERO	Same as above but with equality.
HALT	The program is ended.

Table 1: An example instructions set of a RAM.

Note that the instruction set may be altered according to the desires of the user of the model. Thus an explicit stating of the instruction set is needed for a fully satisfactory description of the machine. We will consider each instruction as belonging to one of two groups: the *arithmetic operations*, the instructions that perform an arithmetic step of the algorithm (in the above instruction set this is

ADD, SUB, MUL and DIV), and the *bookkeeping operations*, that is all other instruction. The name "bookkeeping operation" reflects that these instruction do not perform any mathematics, so to say, but rather organises the memory such that the arithmetic operations can be suitably performed.

The idea is as always to translate the instructions of a program into a time measure, and for the `RAM` model, there are two time-cost functions that we will consider:

- *Uniform cost*: Each instruction of the instruction set costs one time unit.
- *Logarithmic cost*: The time cost of performing an instruction is proportional to the bit-size of the operands, according to the function

$$l(N) := \begin{cases} \log_2(N) + 1, N > 0 \\ 1, otherwise, \end{cases} \tag{11}$$

  where N is an non-negative integer.

For example would multiplying two integers of length N_1 and N_2 respectively take around $l(N_1)+l(N_2)$ time under logarithmic cost, and this might be much more than the 1 time units it would take under uniform cost function.

Additionally, one should note that the `RAM` model does not consider parallelisation, since the cost functions imply that two instructions may not take place at the same point in time. The `RAM` model does however allow for loops and recursions. Finally, one should note that if one uses uniform a time cost function, then the `RAM` model does not take into account that operating with large registers (or many registers) might incurr significant time penalties in reality.

If one ignores the intricates of the model, then one can describe it as

Domain	$\mathbb{Z}$
Instruction set	Flexible, must be specified.
Parallelisation	Not considered
Loops and recursion	Considered
Time cost function	Uniform or logarithmical
Bookkeeping operations incurr costs	Yes
Total memory size affect time directly	No
Memory usage affect time	Yes

Table 2: Description of `RAM` model properites

2.9.3 Straight-line programs

A further restriction (or abstraction, depending on one's point of view) of the `RAM` model is the practice of considering *straight-line programs (SLP)*, which as the name implies consists of a program for, for example, a `RAM` that has been unrolled into a straight line. This unrolling is combined with that the input tape is replaced with specified places in memory from the beginning being assigned to the input values and the output tape has been replaced with that specific registers are assigned to the role of storing the values to be read as the result after the run.

These changes are done with the motivation that it allows one to reduce the instruction set to only the arithmetic operations, for example $\{ADD, SUB, MUL, DIV\}$. The argumentation for why this simplification does not invalidate the resulting model is more or less that the other operations make out a neglible part of the operations performed by the machine, and thus does not affect significantly how well the model represents a realistic computer.

Definition 2.16 (Straight-line program (SLP), adapted from [AHU74])**.**
A straight-line program (SLP) is an ordered set of steps, each denoting the performing of an instruction belonging to a given instruction set consisting of arithmetic operations.

The changes made does result in that programs in the straight line model does not allow for loops or recursions, since these are to be unrolled, and since there are no bookkeeping operations, such as those enabling branching, in the instruction set then there is no cost from using such.

Domain	$\mathbb{Z}$
Instruction set	Flexible, must be specified. Only arithmetic operations.
Parallelisation	Not considered
Loops and recursion	Not considered
Time cost function	Uniform or logarithmical
Bookkeeping operations incurr costs	No
Total memory size affect time directly	No
Memory usage affect time	No

Table 3: Description of straight-line model properites

One should note that since all instructions that exist in this model are the arithmetic ones, then the time cost is to be seen as direct measure of the number of arithmetic operations in the program. However, it is important to note that the choices to be made in the assumption of a specific SLP, such as the exact instruction set or the time cost function, can highly affect the resulting analysis of an algorithm. Thus, whilst a specified SLP does well represent the counting of arithmetic operations, the tradition of sloppily counting arithmetic operations without properly specifying a model of computation (such as the SLP on a `RAM`) does not carry with it the stringency offered by the SLP formalism.

2.9.4 Bitwise straight-line programs (bitwise-SLP)

The SLP convention may be modified by enforcing that registers may only hold single bits, resulting in bitwise-SLPs. This necessarily translates the arithmetic operations into bit operations, such as for example $\{AND, OR, XOR, NAND, NOT\}$.

The motivation for preferring this modelling comes partly from the fact that electronic circuits operate in binary, and thus this convention more closely reflects the performance to be expected from a implementation of the algorithm in a circuit. The model also has the advantage that uniform and logarithmical time cost functions coincide, since all operands have length 1.

In the same manner that the SLP formalism reflects the approach of counting arithmetic operations, the bitwise-SLP reflects the idea of counting bit operations.

Domain	$\mathbb{F}_2$
Instruction set	Flexible, must be specified. Only binary arithmetic operations.
Parallelisation	Not considered
Loops and recursion	Not considered
Time cost function	Uniform
Bookkeeping operations incurr costs	No
Total memory size affect time directly	No
Memory usage affect time	No

Table 4: Description of bitwise SLP model properites

2.10 big-O notation

Let for illustrationary purposes $T(N)$ denote the worst-case time complexity of some algorithm for the input size N. These functions are, as touched upon previously, usually considered in the *asymptotical* sense, meaning that they describe how the function behaves as N goes to infinity. This point of view is chosen for a few reasons, for example to avoid the bias of chosing particular values of N or because obtaining complexity estimates in this framing is easier than finding $T(N)$ explicitly.

There are multiple notions of the asymptotic relationship between two functions and we will use parts of a family of notations commonly called the *Bachman-Landau notation* [Bac04][Lan09].

Definition 2.17 (big-O notation)**.**
Let $f : \mathbb{R} \to \mathbb{R}$, $g : \mathbb{R} \to \mathbb{R}$ *and* $N \in \mathbb{R}$. *Say that* $f(N) = O(g(N))$ *iff*

$$\exists k > 0 : \exists N_0 > 0 : |f(N)| \leq k \cdot |g(N)| \forall N > N_0.$$

Equivalently, one could say that f is upper bounded by g asymptotically, i.e. for sufficiently large N, when ignoring constant factors. Let similarily $f(N) = O^*(g(N))$ and $f(N) = \tilde{O}(g(N))$ denote that f is asymptotically bounded by g whilst suppressing polynomial respectively logarithmic factors in N, i.e $f(N) = O^*(g(N)) \implies f(N) = O(n^c g(N))$ and $f(N) = \tilde{O}(g(n)) \implies f(N) = O(\log_2(N)^c g(N))$ for some $c \geq 0$.

3 Problem formulations

3.1 Problem statement

For all of this thesis, let

$$E := \{P_j(\mathbf{x})\}_{j=1}^{m}$$

be a system of m polynomials of degree at most d in n variables $\mathbf{x}$ that may take on values in $\mathbb{F}_2^n$, where each polynomial may be stated as a sum of monomials

$$P_j(\mathbf{x}) := \sum_{\mathbf{u}\in\mathbb{F}_2^n} \alpha_{j\mathbf{u}} \prod_{i=1}^{n} x_i^{u_i},$$

$\alpha_{j\mathbf{u}} \in \mathbb{F}_2$. Henceforth will we reserve the letter E for denoting the system that is the input to the problem.

Let us call the problem of Polynomial System Solving over $\mathbb{F}_2$ the PoSSo$_2$ problem, and then define our primary view of what it means to solve this problem as:

Definition 3.1 (PoSSo$_2$-Search problem)**.**
Input: *A polynomial system $E \subset \mathbb{F}_2[\mathbf{x}]$*
Output: *One solution, or root, to E, that is, an assignment $\mathbf{x}^* \in \mathbb{F}_2^n$ such that $P_j(\mathbf{x}^*) = 0$ $\forall j \in \{1, ..., m\}$, or $\emptyset$ if no such assignment exists.*

3.1.1 Other problem versions

There are also other notions of what it means to solve the PoSSo$_2$ problem. Below follows a collection of some alternative problem versions:

- *Search*: Find one root to E or declare that no root exists.
- *Decision*: Output a bit describing if there are any roots of E or not. 0 means that there are no roots and 1 means that there is at least one.
- *Exhaustive*: Find all roots to E, or declare that no roots exists.
- *Count*: Output the number of roots to E.
- *ParityCount*: Output the parity of the number of roots to E, i.e. decide whether there are an even or odd number of roots.
- *MultParityCount*: Output a vector of partial parities for a set of subsystems of E, i.e. declare whether each of those subsystems have an even or odd number of roots.

Of particular interest is the decision version of PoSSo for quadratic polynomials. This problem is called $\mathcal{MQ}_2$ and its importance can be motivated by that it is considered the easiest version if one assumes $d > 1$. For all of the versions, the subscript 2 signifies that the problem is over $\mathbb{F}_2$ and thus the index is often omitted in this thesis either when the context is obvious or the size of the specific field does not matter for the given argument.

3.1.2 Reductions between the different problem variations

All the problem variations above are polynomial-time reducible to the search version, meaning that solving any of the other variations yields a solution of the search version of the problem in polynomial time. The formal definitions of the other problem versions and these reductions to PoSSo_2-Search are presented in the appendix together with proof-sketches of the reductions. In Figure 3 we see an illustration of the efficiencies of the reductions, with an arrow from A to B meaning that problem B can be solved using the indicated number of instances of A.

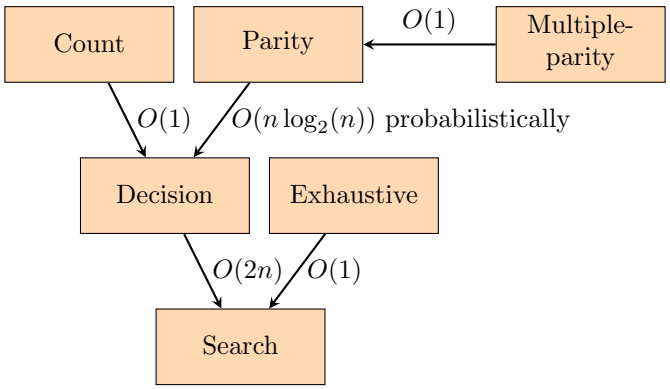

Figure 3: Overview of the reductions between problem versions.

3.1.3 Some reasons to focus on the search version

Our preference to focus on the search version of PoSSo comes from our primary usecase, that is, cryptography. First of all we can note that the usefulness of the decision and counting versions in themselves are quite non-existent for many cryptanalytical applications. Say for example that $\mathbf{x}^*$ corresponds to a given plaintext and $E(\mathbf{x}^*)$ corresponds to the ciphertext, then learning that the ciphertext indeed does correspond to some plaintext does not really help. Similarily, the usefulness of the parity versions in themselves must be seen as limited. The remaining alternative to the search version of PoSSo would then be the exhaustive one. The reasons why the search version then is to be prefered stems partly from the idea that cryptographical schemes tend to be designed such that it should be exceedingly unlikely to find multiple different secret parameters resulting in the same output, a design goal called *second-preimage resistance*, and partly by the remark that should there none the less exist multiple roots, then it is not clear why it would matter much to the adversary which one of them they find.

3.2 Hardness of the problem

The hardness of PoSSo_2-Search is highly dependent upon the choice of the parameters n, m, d describing E. We for a start note that if E is linear, then the problem is solvable in polynomial time by, for example, Gaussian elimination. Interestingly, the problem can be shown to be $\mathcal{NP}$-complete with respect to n already for the case of $d = 2$, that is, for a system of quadratic polynomials. The proof that PoSSo is $\mathcal{NP}$-hard for all but linear systems is based on a reduction from 3-SAT[2] to $\mathcal{MQ}$ [FY79]

[2]3-SAT is the problem of deciding whether or not a given 3-CNF, i.e. a boolean formula of some number of clauses (connected by ANDs) where each clause is a formula of 3 boolean variables (or their negation) connected by ORs, has a satisfying assignment. If one views AND and OR as analog to multiplication and addition respectively, then a boolean formula is a CNF if it is the product of sums of 3 boolean variables (or their negation) [AHU74].

and the realisation that PoSSo is at least as hard as $\mathcal{MQ}$.

Also the relation between m and n is of utmost importance. For the case of very underdetermined systems, $m \ll n$, Kipnis, Patarin and Goubin [KPG99] showed that there are algorithms that solve $\mathcal{MQ}$ in polynomial time and for the very overdetermined case, $m \gg n$, one may solve the problem by simple linearisation and Gaussian elimination. Since the underlying assumptions of these cases cannot be guaranteed generally for the systems ocurring in our area of interest, cryptography, we disregard these cases and focus on the less convenient case where m and n are thought to be somewhat close to each other.

The *exponential time hypothesis*[IP01], which may be seen as an extension of the $\mathcal{P}$-vs-$\mathcal{NP}$ conjecture, says that there is no subexponential algorithm for solving k-SAT, and consequently $\mathcal{MQ}$, in the worst-case. Thus if one assumes the hypothesis to hold then one knows that the complexity of any algorithm solving the PoSSo will asymptotically be of the form $O^*(2^{\delta n})$ in the worst-case. Therefore one might reasonably consider the goal to design an algorithm with a value of δ as low as possible as the primary aim of development. There are however many other goals to thrive for, for example:

- to create an algorithm with a subexponential expected time complexity,
- to create an algorithm with hidden factors as small as possible, or,
- to create an algorithm with an efficient implementation for relevant problem sizes.

4 How to compare?

In the below subchapters we will argue that credibly comparing performance results of algorithms involves quite a few hurdles and that these must be considered to defend the conclusions drawn from the comparisons.

We identify the following areas that ought to be thoroughly pondered:

1. *Inconsistent choices of, and unclearly stated, computational models: A result regarding the performance of an algorithm is only meaningfully comparable if one knows exactly what is measured. Similarily, a comparison between two complexity results are only valid if either they both use the same model of computation or if the relationship between the different used models is known and taken into account. Therefore much care must be taken to either formalise implicitly stated models of computation more and unifying results to the same model, or alternatively argue why one's comparison is essentially agnostic to possible differences in model choice.*
2. *Unclear considerations of space requirements: A comparison of efficiency requires that one states how one will take memory requirements into consideration. This could for example mean to state that one will entirely disregard space, to choose a computational model where space complexity is a part of the time complexity or considering the space complexity separately.*
3. *Difference between expected and worst-case time complexities: Perhaps obviously, one must note that expected and worst-case complexities are not directly comparable, since they fundamentally represent different aspects of the algorithm's behaviour. Therefore it is critical that one remarks suitably upon why one prefers one algorithm above another in the case that their results are given in these two differing measurments.*
4. *Finding a balance between experimental and analytical results: The merits of an algorithm unavoidably relies on the possibility that it could actually be implemented in a computer and executed. It is therefore critical to explain how one weighs already achieved experimental results against hypothetical future experimental results, since these conjectured merits are exactly what analytical results give claims of.*
5. *Defending underlying assumptions: As we will become aware of, it is commonplace to design algorithms that only work (or perform reasonably well) for a subset of all instances of the problem. These restrictions of the intended use-case are done in the form of making assumptions about the input to the algorithm and therefore any comparison between algorithms that make different assumptions about the input space is also fundamentally dependent on the argumentation about why one would accept making such assumptions.*

After these five obstacles have been discussed, we will in Chapter 4.6 give the answer to how we intend to overcome them. This entire chapter is to be seen as a defence of the methodology and reasoning applied in later chapters and, ultimately, also as a defence of the credibility of the conclusions drawn in this thesis.

4.1 A remark on complexity measures

Since the notion of computational time complexity of an algorithm is phrased formally in terms of some specified model of computation, say a `TM` for example, one would be forgiven for thinking that claims of time complexities of algorithms in academic papers necessarily are prepended by a declaration of what model of computation is used. Alas not. As Buillaguet noted in 2022 [Bou22], Diem before him in 2011

[Die11b], Schönhage before him in 1980 [Sch80] and as we will note in the next chapters, there does seem to be a widespread ease with which designers of algorithms takes the rigor of the framing of their complexity analyses. Whilst at times a computational model is stated, it seems to be commonplace to simply state for example "number of bit operations" as the unit of complexity, or even leave entirely to the reader to imagine what the author means by "time complexity". This apparent unimportance of formalism in complexity estimates directly births the questions:

Is it actually needed to formally declare one's used computational model?
Can we not draw detailed, and credible, enough conclusions without them?
and
How does complexity estimates calculated using different models of computation relate to each other?

Concisely, the answer to the first two is

The formal models, or at least descriptions of the capacities one assumes about one's mathematical machine of the same level of detail and unambiguity, are direly needed if one wants to compare complexity results without a much detailed argumentation about the validity of one's chosen methodology for comparing. Credible conclusion can however be drawn, but entirely conditional on the depth of the motivation and the roughness of claims.

The short answer to the third question is

In a somewhat complicated, non-trivial manner.

Longer answers follow below, with an emphasis on why one needs to care about these formalities.

4.2 On counting arithmetic operations

Whilst counting the number of required arithmetic operations is a convenient and intuitive way of describing the amount of computations in an algorithm, there are a few steps that must be considered before this measure is to be seen as a measure of time complexity.

First of all must one make clear what is meant by an "arithmetic operation". Most reasonably would this simply mean the inherent operations to the domain in which the computations are being done and in our case this would mean the binary *field operations* $\{+, \cdot\}$, equivalently thought of as either addition and multiplication or the gates XOR and AND, operating on a pair of bits. This notion of course translates to addition and multiplication of integers (modulo the field size) if the finite field is larger. For the case of larger integers, one could reasonably without overreaching extend this set of two operations to also include minus and integer division, and for the binary case one might perhaps include other gates such as NOT, OR or NAND. One should note, however, that other operations such as vector operations, the addition of two polynomials or evaluating a polynomial are not to be trivially added to this sometimes vague set of "arithmetic operations". This need for care is due to the significant affect these operations may have on the resulting complexity estimates.

In order to use the count of arithmetic operations (let us say one uses the field operations or explicitly makes clear what is an arithmetic operation) as a measure of time complexity, one must also make clear what implicit assumptions this makes about the time cost of the algorithm. It is assumed that:

- *Only arithmetic operations carry costs*: notably does one assume that bookkeeping operations such as branching, copying memory, reading and writing memory is free.
- *The size of the operands does not affect cost*: If the operands are, say, integers, then these may be of arbitrary size without affecting the cost of an operation.
- *The program runs sequentially (is not parallelised)*: Since each operation adds a unit of time, no two operations are performed during the same timestep and thus the program performing the algorithm is not parallelised.

The assumptions of non-parallelisation and costlessness of bookkeeping operations imply that one, in essence, considers a straight-line program. The lack of impact of the operand sizes implies that one assumes a uniform time cost function. All in all, this means that *counting arithmetic operations is essentially just a sloppy way of describing that one analyses a SLP (with uniform cost function) implementation of the algorithm.*

If one accepts this point of view, then in order to turn counting arithmetic operations into a time complexity measure, one finally only needs to consider the specific choice of *instruction set* (define the set of arithmetic operations).

4.3 Equivalence of computational models

A `RAM` and a `TM` can simulate each other in at most a time cost of $O(n^2)$ [AHU74] and thus the adoption of a `RAM` model does not change the fundamental complexity class of the underlying problem, at least not in the relationship between $\mathcal{P}$ and $\mathcal{NP}$. As for the relationship between a SLP and an unbounded `RAM`, the difference between the models does seem to be at least polynomial in the worst case, for example does multiplying two N-bit numbers take $O(1)$ steps in the `RAM` model but at least $O(N \log_2(N) \log_2(\log_2(N)))$ steps in the bitwise-SLP model [AHU74].

The key take away from this subchapter is that the computational models involved in the upcoming survey are not necessarily equivalent when it comes to speed, and that one therefore needs to be clear about how one is going to take these differences into account. Hopefully the reader is also convinced that whilst the relationship between complexity results given in different computational models is somewhat cumbersome to analyse, the overarching merits of an algorithm are to be seen as likely to shine through (as long as the models are somewhat similar) and that one therefore is able to make at least some comparisons between algorithms that do not have their complexity analyses in the same model.

4.4 The consideration of space complexities

Firstly, let us remark that the decision on how one should weigh space complexity into the idea of the efficiency of an algorithm boils down to the discussion of choosing a computational model. There are a few conflicting interests in this choice, perhaps primarily between the desire that the model should be easy to use and understand and the desire to have it closely resemble the characteristics of an actual physical computer. For the sake of how space complexity affects time complexity, then the most convenient and unrealistic choice is to say that the memory size does not at all directly affect the runtime. This stance is taken by all of the models described in Chapter 2.9 and this approach is commonly used, but also criticised, in the literature.

The debate about the most reasonable computational model for algorithm design and cryptanalysis is,

however, not finished. This can, for example, be seen in that Bouillaguet during the time of us writing this thesis released a paper [Bou22] outlining some prominent positions on the topic, and said paper is to be seen as an attempt at bringing wood to the fire of the debate. One key idea to ponder, perhaps, from that paper is whether the lack of cost for memory in the `RAM` model might lead to the designing of attacks that incur a vast increase in memory usage in order to decrease the analytical runtime and thus whilst they look highly efficient theoretically, actually in reality are unimplementable due to the memory demands.

If one chooses a computational model that does not factor space requirements into the time complexity, then one realises that one must separately regard both time and space requirements in order to assess the efficiency of an algorithm. There are a few different ways of doing this, explicitly or implicitly, such as saying that time complexity is of primary importance and space complexity is only a tie-breaker or the other way around. Finally, one could choose to regard the space requirements as an assumption made by the algorithm, or as a condition in the complexity results. Complexity claims would then go along the line of "the time complexity of the algorithm is $O(T(N))$ in the `RAM` model, *under the assumption that the machine has access to a memory, that is capable of instantaneous accesses, of size* $O(S(N))$" (where T and S are some functions giving the claims of the time and space complexities respectively).

4.5 Practical versus analytical results

Analytical results of the complexity of an algorithm carry importance purely due to the fact that they are to be seen as indicators of the efficiency of future implementations of said algorithm. It could therefore be argued that analytical results are to be viewed as less important than experimental results, since the analytical results are simply hypotheses of future experimental merits. On the other hand must one note that analytical results have certain advantages of paramount importance, not the least that they avoid the bias of all the little choices involved in implementing an algorithm (such as choosing hardware, programming language, data structures etc.) and also that they allow for understanding the performance of an algorithm for problem sizes that are out of range for current computers.

Due to the considerable merit in both of these two lines of argument, it becomes clear that one needs to take into account both analytical and experimental results when one compares the performance of algorithms. Especially, reasoning behind the extendability of a result of one type into one of the other is highly valuable. For example should one reasonable lay more weight on an analytical result if there are indications that its predictions correspond well to available experimental data and one should similarily lay more weight on an experimental result if its data is well understood analytically.

4.6 How we intend to compare

In this subchapter we declare how we intend to clear out the issues presented at the start of this chapter.

Inconsistent choices of, and unclearly stated, computational models
We will not translate any complexity results from one model to another, or from an implicitly stated model to an explicit one. This is due to the time restrictions of this project and due to the fact that the time needed to sufficiently understand the implicit assumptions in some of the complexity analyses is significant. We will rather avoid the need for direct comparisons between complexity results by arguing that many algorithms may be compared through the fact that they are based on each other and that therefore their complexity analyses directly relate. We will thus exclude many methods through "redundancy arguments" and the remaining methods will be compared according to what

properties of the input system let them perform particularily well, thus being able to partition the input space roughly into subspaces where we may conjecture that some specific algorithm is likely the most efficient.

Space complexity
We will *regard space complexities separately from time complexities*, and split the judgement of the state-of-the-art according to whether or not one is willing to ignore space requirements. This means more exactly that we will treat the access to a machine with memory size exponential in the input size as an assumption on all complexity results and then split the conclusions of the current state-of-the-art by whether or not an algorithm needs this assumption or not.

Worst-case versus expected complexity
Naturally, we see worst-case results as more important than those of expected complexity if the complexities are somewhat similar. However, when a significantly lower expected complexity for one algorithm is compared with a higher worst-case complexity of another algorithm, we will consider the algorithm with the expected complexity as more efficient. This judgement is, however, dependent on that the statistical assumptions required to reach the expected complexity result are highly defendable for the use-case of cryptography.

Experimental vs analytical results
We will give primary importance to analytical asymptotical time complexity results and factor in *experimental results as possible tie-breakers* between methods of similar analytical performance. The reasoning behind this is that the range of problem sizes that are of greatest importance for cryptography tend to lie vastly out of reach for modern implementations. Significant weight is put on concerns that some analytical results are unlikely to be faithfully reproduced in practice, and this mostly takes the shape of discussions about memory requirements.

Defending underlying assumptions
We will disregard methods requiring assumptions that restricts the input system too much. Assumptions deemed as weaker restrictions are however allowed and are remarked upon in the conclusions of the state-of-the-art. A more satisfyingly detailed discussion on the topic follows in Chapter 5.

Additional note - Ignored terms in big-O notation
The comparative merit of results given in the notations O, $\tilde{O}$ and O^* boils down to the intended use of the results. Since the primary intended use-case of the algorithms surveyed in this thesis is cryptanalysis and one therefore is interested primarily in explicit time estimates, the size of hidden factors of the notation becomes critical. A common line of argument goes roughly as *"the complexity estimate is $O(2^N)$ but since O has been experimentally shown to only hide small factors and terms, we ignore the hidden terms and conjecture that the algorithm actually in essence takes 2^{N^*} time for some given N^* of our choice"* (where N is the variable describing the input size of the problem and N^* is some fixed input size). Whilst lines of reasoning similar to this one are inherently rough and approximative, we would argue that it is defendable for O if the experimental results are sufficently strong but that ignoring the posibility of large hidden factors in $\tilde{O}$ or O^* must be seen as grasping for too much. Therefore the usefulness of results given in such looser notations are, we claim, more or less only of theoretical scholarly interest. This somewhat harsh stance does, however, not mean that algorithms without more detailed asymptotic complexity results are to be disregarded. It does rather mean that experimental results are granted a larger emphasis in the judgement of said algorithms.

5 Case specification

As a consequence of the vast amount of research done on the subject of solving PoSSo, there have been many algorithms developed that are specialised for some specific case of PoSSo. In this chapter we will declare more thoroughly what case is considered in the upcoming literature survey, motivate why we have chosen to specifically consider this point of view and briefly give an introduction to some main results in other cases.

First of all, we have chosen to *only consider polynomials over* $\mathbb{F}_2$. Why this field is to be regarded as a finite field of particular importance was outlined in the introduction and will not be elaborated on further. It is however interesting to reason shortly about why one would like to restrict the survey to some given field at all. The given time constraints of the project dictate that not all especially important subcases of PoSSo can be discussed at length and therefore one might ponder why we have chosen to concentrate entirely on the binary field instead of writing the survey on the methods available for an arbitrary field and then regard methods only working in $\mathbb{F}_2$ as a footnote, similarily to how we treat $\mathcal{MQ}$ or very overdetermined systems. The reason for taking our chosen path instead of that one is primarily due to that much of the important advances in the later years, such as the introduction of the algorithms `FES`[Bou+10], `Crossbred`[JV18] and $\texttt{Dinur}_2$[Din21], all work only over the binary field and thus this case is in more dire need of surveying than the general one. Also, the reason for why the above mentioned algorithms demand input over $\mathbb{F}_2$ is intellectually quite rich and is explored more thoroughly in Chapter 13.3.

We have further chosen to *consider polynomials of arbitrary degree* d. Similarily to in the paragraph above, there is also here a quite tempting alternative route, namely that of restricting the survey to methods for $\mathcal{MQ}$ in order to free up space and time for digging more satisfyingly into subcases such as very over- or underdetermined systems. The reason for not taking this alternative route was that we deemed the restriction on the direct applicability of conclusions drawn in this report as being more of a loss than the gain of better understanding additional subcases. This can also be formulated as that we deemed the case of non-quadratic systems more interesting and significant than the case of very over- or underdetermined systems.

We *do not consider any quantum algorithms*. Whilst there is some literature on quantum algorithms (for example [Fau+17]) for solving PoSSo and these very much are of long-term theoretical interest, it was deemed that the new traditional algorithms are of more direct need for being widely understood. Additionally, it was decided that the overhead of needed time to satisfyingly remark on available quantum algorithm was too high in relation to the time frame of this thesis.

We assume that E *is neither very overdetermined or very underdetermined* and *is to be seen as essentially random*. These assumptions are both made since such instances of PoSSo are believed to be the hardest to solve and thus these assumptions more or less simply say that we will not deem an algorithm as being state-of-the-art if it makes highly restrictive assumptions about the structure of the system or demands that the system is very over- or underdetermined. The assumption that the system may well be assumed to be practically indistiguishable from one generated uniformly at random also stems from the idea that this is an idealistic view on systems occurring in cryptography and thus attacks performed under this assumption may be seen as conservative estimations of the attack's practical threat.

Similarily stemming from the use-case of cryptography, we assume that *there is a small, positive number of roots* of the system. This is because we assume that the systems considered actually comes from a faithful representation (and execution) of a cryptographical construction. For example, in the

setting described in Chapter 1, we will assume that the encrypted message that the adversary tries to decrypt (by solving a polynomial system) is generated by real encrypted communication and thus it follows that there is an original message that corresponds to the encrypted one and this original message is represented as a root of the polynomial system.

5.1 The setting

To summarise; Unless otherwise explicitly stated, the setting considered is as follows:

- We regard the PoSSo_2-Search problem.
- We assume that m and n are of essentially equal size.
- Assume $d \geq 2$.
- We assume that the number of roots to E is positive and small.
- We make no further assumptions on the properties of E, and may thus suitably consider it in essence indistiguishable from a uniformly random system of polynomials.

5.2 Important cases outside of our scope

5.2.1 Very overdetermined systems

When it comes to very over- or underdetermined systems, the aim is to utilise either the extra information in the vast number of equations or the extra flexibility given by the high number of variables. For the overdetermined case, this fundamentally takes the shape of *linearisation* (see, for example, [Cou+00]), meaning that one turns the nonlinear system E into a linear system E' that can then be solved in polynomial time using, for example, standard Gaussian elimination.

Linearisation works by regarding each unique monomial of degree above 1 as a new variable, and possibly introducing equations relating these new variables. The need for having a lot more equations than variables in the original system is that the equations must be able to "interpret" these new variables in a detailed enough way such that the number of solutions to the linearised system is low enough for them all to be tested in polynomial time. It is also clear that since linearly dependent polynomials do not contribute any new information to the system, the critical demand is the number of linearly independent polynomials after the linearisation.

In order to be able to reach somewhat general results, one assumes that the system E is uniformly randomly generated. Then one can note that for an underdetermined linearised system E', the likelihood of it having an exponential number of *parasitical solutions*, i.e. roots that are not roots of E, is high but for a system E' that is at least determined, then one would expect it to have a unique solution and that this solution also leads to a solution of E [KS99]. Thus the efficiency of linearisation is dependent on how the number of new variables relate to the number of equations.

More concretely does this mean that as long as m is larger than the total number of variables after linearisation, we expect the resulting linear system both to be solvable in polynomial time, and the number of parasitical solutions to be polynomially many. Thus, since the linearisation in itself requires only polynomial time, then the entire procedure takes *expectedly* polynomial time (in for example the `RAM` model) if the original system E is overdetermined enough and in essence such as if it was uniformly

randomly generated. The number of variables after linearisation for a system of polynomials in degree at most d in n variables is

$$\sum_{i=1}^{d} \binom{n}{d}.$$

This means for example that a homogeneous quadratic system can be solved in expected polynomial time as long as $m \geq \frac{n(n+1)}{2}$ (see, for example, [Cou+00]).

Linearisation is, to the best of our knowledge, folklore. In 1999, Kipnis and Shamir[KS99] introduced *relinearisation*, a technique that improves linearisation by noting that the newly introduced variables are in fact not unrelated to each other, and thus new non-trivial equations describing how they relate to each other can be added to the linear system and thereby decrease the demand of how overdetermined E must be. This relinearisation technique is improved further in 2000 by Courtois, Klimov, Patarin and Shamir when they introduce the *eXtended Linearisation (XL)* algorithm [Cou+00]. This algorithm is able to solve $\mathcal{MQ}$ in expected $n^{O(\frac{1}{\sqrt{\epsilon}})}$ time if the $\mathcal{MQ}$ instance has $m \geq \epsilon n^2$, for some $0 < \epsilon \leq \frac{1}{2}$.

5.2.2 Very underdetermined systems

Somewhat similarily to the approach above, the idea for very underdetermined systems is to transform E into a linear system E' which is then solved in polynomial time. Also as above, this is achieved by a variable substitution. This time however, the demand for a large number of variables stems from the need to make sure that there is a solution to the linear system rather than avoiding to have too many parasitic solutions, as it was above.

Through similar argumentation as for the previous case, the result is reached that as long as the system is sufficiently underdetermined and in essence such as if it was unformly randomly generated, then it can be solved in *expected* polynomial time. This was first shown by Kipnis, Patarin and Goubin in 1999 [KPG99], and then the demand was for $\mathcal{MQ}$ systems that $n \geq m(m+1)$. This requirement has since then been lowered in subsequent works, for example did Cheng et al. [Che+14] give an algorithm that for $\mathcal{MQ}$ only demands that $n \geq \frac{m(m+1)}{2}$.

6 Survey

6.1 Overview of literature study

In the upcoming four chapers follows a review of the research articles constituting the state-of-the-art of algorithms for solving PoSSo_2. We start by considering the approach of exhaustive search, that is, simply trying all possible inputs to E to test if any of them is a root. Following this perspective the algorithm *fast exhaustive search* (`FES`)[Bou+10] is presented. After that, a recurring theme through the rest of the literature will be that the algorithm designers try to reach an asymptotic complexity better than that of exhaustive search. This is often done focusing heavily on asymptotic complexity by ignoring polynomial or logarithmic factors, or introducing new assumptions about the problem instance to better exploit the structure arising from said assumptions.

After the exploration of exhaustive search, we consider older, more purely algebraic approaches, of which the most significant ones are based on finding a *Gröbner base* of E. Whilst these methods have a horrible (doubly exponential) asymptotic worst-case complexity in general, they do perform vastly better for so-called *semi-regular sequences*. These methods do also have the advantage that they are general in the sense that they do not rely on acting specificly on $\mathbb{F}_2$. Algorithms that fall in this category is F_4, F_5, `XL` and their continuations [Fau99][Fau02][BFP09] [Cou+00][Din+08].

Then we review methods that utilise a wide range of ideas stemming from previous works, such as *guessing variables*, *solving sparse matrices* and *exhaustive search*. Whilst there are much improvements made in these methods, the overarching ideas remain quite the same and thus they are conveniently considered in group. Algorithms that fall in this category are `Crossbred`, `BooleanSolve` and `BDT` [JV18][Bar+13] [BDT21].

Finally, we consider algorithms utilising the recently introduced *polynomial method*, which is more or less centered around representing the entire polynomial system in a single polynomial and then evaluating that polynomial probabilistically. These methods are of particular interest since they are the first ones that achieve an exponential speedup over exhaustive search without making major assumptions about the problem instance. Algorithms that belong to this family are the algorithms of Lokshtanov et al., Björklund et al. and the two algorithms introduced by Dinur in 2021 [Lok+] [BKW19][Din][Din21]. In Table 5 we present a timeline of some imporant developments that have lead to the current state-of-the-art.

Let us also remark upon some notation and stylistic choices made for the upcoming literature review; Firstly, note that the notation used in the literature is by no means consistent, perhaps as a consequence of the wide range of mathematical disciplines and traditions that have contributed with ideas over many years of research. Primarily we have chosen to adopt the notation style used by Dinur [Din][Din21] since these papers birthed the idea of this thesis topic. Secondly, we should say that most subchapters starts with a short presentation of an idea that we have found helpful for framing an intuition about the algorithm in question. The formulations of these key ideas are in most cases done by us and are meant to give the reader a feeling of what might have motivated the development to go in that specific direction. They are however not meant to imply that specific idea was necessarily original to the authors of the corresponding paper and neither that it is necessarily the most important academic contribution offered by the paper.

One should also bear in mind that most algorithms that we give in pseudocode are restructured compared to the papers they are based upon. This is primarily done by renaming and reformulating subprocedures so that there is a larger similarity between the algorithms surveyed. The hope is that

Year	Development
1965	Buchberger introduces the notion of *Gröbner bases* and an algorithm for calculating them.
1993	Faugére et al. gives an improved algorithm, *FGLM*, for changing the underlying ordering of a Gröbner basis.
1999	Faugére presents F_4, an improvement of Buchberger's algorithm.
2000	The *XL algorithm* is introduced by Courtois et al. Severly overdetermined systems may now be solved in polynomial time.
2002	Faugére presents F_5.
2004	Ars et al. proves that `XL` is a redundant version of F_4.
2010	Bouillaguet et al. introduces *Fast exhaustive search (**FES**)*. This is lowers the expected time complexity of exhaustive search from $O(m2^n)$ to $O(dlog(n)2^n)$.
2013	Bardet et al. introduces *BooleanSolve*, the first algorithm for $\mathcal{MQ}$ with an expected exponential speedup over exhaustive search, under some general algebraic assumptions.
2014	Williams proposes that *the polynomial method* may be used for cryptanalytical purposes.
2017	Vitse and Joux presents the *Crossbred algorithm*, which now holds the record in parts of the Fukuoka $\mathcal{MQ}$ challenge [Yas+15].
2017	Lokshtanov et al. gives the first algorithm using the polynomial method for solving PoSSo. This is the first algorithm exponentially faster than exhaustive search in the worst-case and it has a complexity of $O^*(2^{(1-\frac{1}{5d})n})$.
2019	Björklund et al. continues the work of Lokshtanov et al. and brings the complexity to $O^*(2^{(1-\frac{1}{2.7d})n})$.
2021	Dinur proposes two algorithms utilising the polynomial method, one that achieves a worst-case runtime of $O^*(2^{(1-\frac{1}{2d})n})$ and one with an expected runtime of $O(n^2 \cdot 2^{(1-\frac{1}{2.7d})n})$. The first is the algorithm to date with the largest worst-case exponential speedup over exhaustive search and the second is the first that achieves an exponential speedup over `FES` whilst still having a known complexity that does not ignore polynomial factors.

Table 5: A timeline over the developments of methods for solving PoSSo_2.

this should help the reader better understand how the algorithms relate whilst still retaining a correct representation of the procedures. Also, we have opted to sometimes ommit giving pseudocode for some subprocedures. When this is done the reader is referred to the original work, and may in the meanwhile assume that the subfunction is calculated directly.

7 Exhaustive search

As a first approach to give a baseline for comparisons, one can note that PoSSo_2 can be solved naively through exhaustive search in $O(m2^n)$ evaluations of the polynomials. This translates to about $O(m\binom{n}{d}2^n)$ bit operations, as shown by Bouillaguet et al. [Bou+10]. It is however possible to do exhaustive search significantly faster.

7.1 Fast Exhaustive Search (FES), 2010

Correctness assumptions:	*None*
Assumptions for complexity:	Uniform randomness of system
Expected time complexity:	$O(d\log_2(n)2^n)$ [Bit operations]
Space complexity:	$O(mn^{d-1})$ [Bits]

Idea: *Perform exhaustive search on the full domain $\mathbb{F}_2^n$, enumerated efficiently, for only some of the polynomials, then search only amongst their common roots for the rest of the system!*

In 2010, Bouillaguet et al.[Bou+10] introduces an improved exhaustive search algorithm that solves the exhaustive version of PoSSo_2 with an *expected* complexity of $O(d \cdot \log_2(n)2^n)$ bit operations. The algorithm is called *fast exhaustive search (**FES**)* and is, in essence, summarised by the idea at the start of this subchapter. The reasoning goes roughly like this:

1. A single polynomial can be exhaustively searched on $\mathbb{F}_2^n$ using at most $O(d2^n)$ bit operations.
2. Choose some $c \in \{1, ..., m\}$ and solve the first c polynomials as above. This takes in total at most around $O(cd2^n)$ bit operations.
3. Then if one assumes that there are in expectation 2^{n-c} common roots resulting from the previous step, then one may search the remaining $m-c$ polynomials naively on those roots in $O(\binom{n}{d}2^{n-c})$ bit operations.
4. The total runtime of the algorithm is then approximately $O(cd2^n + \binom{n}{d}2^{n-c})$ and if one balances these terms then the *expected asymptotic* time complexity becomes

$$O(d\log_2(n)2^n). \tag{12}$$

Whilst perhaps conceptually simple, it is not necessarily that simple to see why each of the claims above holds. Therefore let us explore them further below.

Claim 1: A single polynomial can be exhaustively searched on $\mathbb{F}_2^n$ using at most about $O(d2^n)$ bit operations.
This claim is achieved by noting first that the entire $\mathbb{F}_2^n$ can be enumerated such that subsequent elements only differ in one bit by utilising a so called *Gray code*. Let $\oplus$ denote the bitwise XOR operation (over $\mathbb{F}_2^n$) and $\ggg$ denote the logical right shift (the bit string, the left side operand, is

shifted to the right some number of steps, given by the right hand operand, and the vacant positions are filled with zeros). Then the Gray code is defined by

$$GrayCode(\mathbf{i}) := \mathbf{i} \oplus (\mathbf{i} \ggg 1), \tag{13}$$

where $\mathbf{i} \in \mathbb{F}_2^n$ is an integer represented as a bitstring. Equivalently to this definition of the Gray code, one may define it (as it was in the 1947 patent by Frank Gray [Gra47]) through that the entire sequence of the Gray code for n bits can be constructed as such:

1. Write the first two numbers in their usual order, that is
$$\{0, 1\}.$$
2. Append their reflection, i.e.
$$\{0, 1, 1, 0\}.$$
3. To the first half of the list, add a 0 before all elements, and add a 1 before thos in the second half, i.e.
$$\{00, 01, 11, 10\}.$$
Now one may note that this sequence satisfies the properties we have described that a Gray code of two bits should have.
4. To extend this 2-bit code into one of three bits, the same "reflect-and-append" trick is used again, resulting in
$$\{000, 001, 011, 010, 110, 111, 101, 100\},$$
which, as desired, is a 3-bit Gray code sequence.
5. This may be repeated to arrive at a full Gray code sequence of arbitrarily many bits.

From this second (equivalent) definition of the Gray code, it becomes clear (through induction) that it indeed does have the two desired properties, namely that it iterates over the entire domain and that subsequent elements differ in only one bit. Thus it may be used as a transition function through noting that

$$GrayCode(\mathbf{i} + 1) = GrayCode(\mathbf{i}) \oplus e_{b_1(\mathbf{i}+1)}, \tag{14}$$

where e_j is a canonical basis vector and $b_1(\mathbf{i})$ is defined as the index of the least significant bit in $\mathbf{i}$ that is set to 1. Therefore one may enumerate the entire domain by setting $\mathbf{x}^{(0)} := GrayCode(0) = 0$ and then $\mathbf{x}^{(i+1)} := \mathbf{x}^{(i)} \oplus e_{b_1(\mathbf{i}+1)}$.

Also define the *boolean derivative* $\frac{\delta P}{\delta x_i}$ of a polynomial $P \in \mathbb{F}_2[\mathbf{x}]$ with respect to x_i as

$$\frac{\delta P}{\delta x_i}(\mathbf{x}) := P(\mathbf{x} + \mathbf{e_i}) + P(\mathbf{x}), \tag{15}$$

where $\mathbf{e_i}$ is the assignment of $\mathbf{x}$ such that only x_i is 1. Then we have that since the Gray code ensures that only one bit changes between assignments of $\mathbf{x}$, then subsequent evaluations of P consists purely of calculating $\frac{\delta P}{\delta x_i}$. The argumentation then goes that since each $\frac{\delta P}{\delta x_i}$ is a polynomial of strictly lower degree than P, then one can calculate it recursively and since that polynomial is agnostic to a specific assignment to $\mathbf{x}$ then all of the derivatives may be computed before the enumeration starts. Thus, if the derivatives are already known, on may calculate the transition between $P(\mathbf{x})$ and $P(\mathbf{x} + \mathbf{e_i})$ in

essence in $O(d)$ bit operations. Finally since this needs to be done 2^n times and initalising the enumeration takes time polynomial in n, then one can find all roots of one polynomial in $O(d2^n)$ bit operations.

Claim 2: All common roots of c polynomials can be found using $O(cd2^n)$ bit operations. The only trouble of turning claim 1 into claim 2 is the need to only output the roots that are common for all of the polynomials. The authors of [Bou+10] proposes that one simply runs c instances of the procedure above in parallel but that at each iteration one checks that the current assignment satisfies all polynomials rather than only one. This check can be done with at most negligible extra time cost and thus the worst-case complexity of finding all common roots of c polynomials is c times the time from the procedure in claim 1, that is $c \cdot O(d2^n) = O(cd2^n)$.

Claim 3: If there is 2^{n-c} common roots of the first c polynomials, then from those roots the common roots of the whole system can be found in at most $O(\binom{n}{d}2^{n-c})$ bit operations. Naively evaluating the $(i+1)$th polynomial on the common roots of the first i does under the assumption take in essence $\binom{n}{d}2^{n-i-1}$ bit operations. Thus one needs an expected

$$\sum_{i=c+1}^{n} \binom{n}{d} 2^{n-i-1} = \binom{n}{d} 2^{n-1} \sum_{i=c+1}^{n} 2^{-i} = \binom{n}{d} (2^{n-c-1} - 2^{-1}) \approx \binom{n}{d} 2^{n-c}$$

bit operations to find the common roots of $\{P_{c+1}, ..., P_m\}$ from the common roots of the first c polynomials.

Claim 4: The total expected asymptotic runtime of `FES` is then $O(d \log_2(n) 2^n)$
The total expected runtime of `FES` is, from the previous claims, obviously $O(cd2^n + \binom{n}{d}2^{n-c})$. Since the only variable in that expression that may be chosen is c, then the quest for the optimal runtime comes down to choosing a c such that the two terms are equal as n grows indefinitely. The goal is thus to solve

$$cd2^n = \binom{n}{d} 2^{n-c}$$

and Buillaguet et al. argues that as n grows, the equation above gives that

$$c \approx \log_2(n)$$

and thus the total expected runtime of `FES` is $O(d \log_2(n) 2^n)$, just as claimed.

In addition to their theoretical results, the authors of [Bou+10] also provide a highly optimised public implementation of `FES` for GPUs. This implementation serves both as a useful tool in practice and as a witness of the influence of designing highly parallelisable algorithms.

8 Gröbner basis based methods

Idea: *transform the system E into a different system of polynomials that has the same roots but is easier to solve!*

The use of of Gröbner bases for solving polynomial systems is outlined in Chapter 2.6. In short, solving polynomial systems by using Gröbner bases typically consist of three distinct steps:

1. Find a Gröbner basis of E for some monomial ordering, preferably grevlex, using for example Buchberger's algorithm, F_4, F_5 or some version thereof [Buc06][Fau99][Fau02][BFP09].
2. Translate the Gröbner basis in grevlex to a Gröbner basis under some other monomial ordering, prefferably lex, by use of for example the FGLM algorithm[Fau+93].
3. Iteratively calculate a root of E from the Gröbner basis under lex, as described in Chapter 2.6.1.

There are a few reasons why one should regard these steps separately rather than together as one algorithm. First of all, there may be different uses of a Gröbner basis than finding the roots, and thus splitting up step 1 and 3 seems reasonable. Secondly, the different desired properties of the monomial ordering in step 1 and 3, together with that there is an efficient algorithm for changing ordering, have lead to heuristical analyses showing that inserting step 2 is more performant than running step 1 under lex.

A natural question that arises from this methodology is the question about which step that demands the main portion of runtime. As we will bear witness to in the upcoming chapters, estimating the runtime of the Gröbner basis algorithms is quite an intricate matter and therefore we will discuss the complexities of both step 1 and 2 (step 3 takes negligible time). In general though, for the comparisons between this approach and other methods for solving PoSSo, we will make the assumption that it is step 1 that dominates the runtime of this method. There are good reasons for this, such as that the complexity of FGLM depends heavily on *the degree of the ideal*, that is the dimension of $\mathbb{F}_2[\mathbf{x}]/\mathcal{I}$, and this one may reasonably believe to be small for systems arising from cryptography. One should however bear in mind that ignoring the complexity of step 2 is a simplification that might not always be so easily defended.

8.1 F_4, 1999

Correctness assumptions:	*None*
Worst-case time complexity:	$O(2^{2^n})$ [Arithmetic operations]
Space complexity:	*Unknown*
Note:	Outputs a Gröbner basis, not roots

Idea: *By replacing the iterative nature of Buchberger's algorithm by one based on reducing matrices, you can somewhat avoid crucial choices of strategy and also achieve a highly parallelisable algorithm.*

In 1999, Faugére proposed his *algorithm number 4*, F_4[Fau99], which is to be seen as an improvement of Buchberger's algorithm (see Algorithm 1) from 1965 [Buc06]. The primary intuition behind F_4 is

that there are several choices to be made during Buchberger's algorithm that, whilst non-critical for correctness, does highly influence the performance. One of these choices concerns how the selection of the pairs of polynomials for consideration is done. Faugére's answer to this choice is, as he frames it, that *"we make no choice"* (see [Fau99], p.5). This avoidance of choosing a clear strategy for selecting individual pairs of S-polynomials is done by selecting a set of polynomials at a time, and then utilising linear algebra techniques to consider a wider range of polynomials at the same time. Another improvement of F_4 compared to Buchberger's algorithm is that the switch to a linear algebra point of view enables a higher degree of parallelism, something that is difficult in Buchberger's algorithm due to its sequential nature.

Definition 8.1 (Critical pair, reducible, top reducible, reformulation of Chapter 2.1 in [Fau99])**.** *Let $\mathbb{F}$ be a field and $\mathbb{F}[\mathbf{x}]$ the polynomial ring in $\mathbf{x}$. Denote by M the set of all monomials in the variables in $\mathbf{x}$, and let $M(E)$ be the set of all monomials in some sequence of polynomials $E := \{P_1, ..., P_m\}$. Order M by some chosen monomial ordering $\succeq$.*

Define a critical pair, or just pair when the context is clear, of two polynomials f_i, f_j as an element $Pair(f_i, f_j) := (lcm_{ij}, t_i, f_i, t_j, f_j) \in M^2 \times \mathbb{F}[\mathbf{x}] \times M \times \mathbb{F}[\mathbf{x}]$ such that

$$lcm(Pair(f_i, f_j)) = lcm_{ij} = LT(t_i f_i) = LT(t_j f_j) = lcm(LT(f_i), LT(f_j)),$$

where t_i, t_j are some two monomials such that the equation above holds. Define also two projections $Left(Pair(f_i, f_j)) := (t_i, f_i)$ and $Right(Pair(f_i, f_j)) := (t_j, f_j)$. We define the degree of a pair p_{ij} as $deg(lcm_{ij})$.

For $f, g, h \in \mathbb{F}[\mathbf{x}]$, say that f reduces to h modulo g if there exists some $t \in M(f), s \in M$ such that $s \cdot LT(g) = t$ and $h = f - \frac{a}{LC(g)} \cdot s \cdot g$, where a is the coefficient of t in f. We say that f is top reducible modulo $G \subset \mathbb{F}[\mathbf{x}]$ if there exists some $g \in G, h \in \mathbb{F}[\mathbf{x}]$ such that f reduces to h modulo g and $t = LM(f)$.

The idea at the start of this subchapter tells us that Faugére represents polynomial systems as matrices in F_4, and the representation he uses is reminiscent to, but different from, the construction of Macaulay matrices that we introduced in Chapter 2.7. Faugére constructs the matrix representing a polynomial system $\{P_1, ..., P_l\}$ by letting the rows be indexed by the polynomials and the columns by all monomials in $\mathbb{F}_2[\mathbf{x}]$, under some monomial ordering. Then the elements in the matrix are the coefficient of the polynomial indexing the row and the monomial indexing the column, that is if we let A be the matrix that represents E under M ordered by some ordering $\succeq$, then

$$A_{i,j} := coeff(P_i, M_{\succeq}[j]), \tag{16}$$

where $coeff(P_i, M_{\succeq}[j])$ is the coefficient in P_i of the jth monomial in M ordered by $\succeq$. Faugére gives no specific name for this matrix representation, so we will call it *the simple matrix representation* of a polynomial system. One may remark that the differences between this representation and that of a Macaulay matrix is that this one does not enforce an upper degree on the polynomials and it does not generate extra rows by multiplying polynomials and monomials as long as the degree is admissable.

Now we have all of the definitions needed to present a basic version of F_4 below in Algorithm 2. The formulations used is a mix between the ones in [Fau99] and [Ars+04] and have also been restructured to allow for more convenient comparisons.

Algorithm 2 Basic F_4

Require: A set $E := \{P_1, ..., P_m\}$ of polynomials.
$G \leftarrow E$, $c \leftarrow 0$
$Pairs \leftarrow \{Pair(P_i, P_j) | P_i, P_j \in E : i \neq j\}$
while $Pairs \neq \emptyset$ **do**
    $c \leftarrow c + 1$
    $A, Pairs_c \leftarrow GenMatrix_{F_4}(Pairs)$
    $Pairs \leftarrow Pairs \setminus Pairs_c$
    $\tilde{A} \leftarrow Reduce(A, G)$
    for $h \in \tilde{A}$ **do**
        $Pairs \leftarrow Pairs \cup \{Pair(h, g) | g \in G\}$
        $G \leftarrow G \cup \{h\}$
    end for
end while
Output: Gröbner basis G of $\mathcal{I} := \langle P_1, ..., P_m \rangle$

Algorithm 3 $GenMatrix_{F_4}$

Require: A set $Pairs$ of critical pairs. A set G of polynomials.
$d = min\{deg(p) | p \in Pairs\}$
$Pairs_{used} \leftarrow \{p \in Pairs | deg(p) = d\}$
$A \leftarrow \{tg | (t, g) \in Left(Pairs_{used}) \cup Right(Pairs_{used})\}$
$Done \leftarrow LT(A)$
while $M(A) \neq Done$ **do**
    $m \leftarrow SomeElementOf(M(A) \setminus Done)$
    $Done \leftarrow Done \cup \{m\}$
    if m top reducible mod G **then**
        $m = m'HT(g)$ for some $m' \in M, g \in G$
        $A \leftarrow A \cup \{m'g\}$
    end if
end while
Return: $A, Pairs_{used}$

Algorithm 4 *Reduce*

Require: A set A of polynomials.
$\tilde{A} \leftarrow RowEchelonForm_{\succeq}(A)$
$\tilde{A} \leftarrow \{f \in \tilde{A} | LT(f) \notin LT(A)\}$
Return: $\tilde{A}$

What is done in F_4 is in essence that matrices representing (using the simple matrix representation) larger and larger degree polynomials are built and then reduced to row echelon form until all polynomials constructed by the linear algebra step are linear combinations of polynomials in the iteratively constructed Gröbner basis. Compared to Buchberger's algorithm, the reductions to normal form are now done for multiple pairs at a time during the matrix reductions to row echelon form and the strategy for selecting pairs is now replaced with a selection strategy for choosing sets of pairs. Choosing such a strategy is however still highly influential on the performance of the algorithm. Faugére proposes the usage of what he calls *the normal strategy*, namely that in each iteration, one selects all the pairs such that their degree is the smallest in the set of remaining pairs.

Remark 1.
One may note that the F_4 algorithm thus itself restricts the degree of the polynomials in the matrix and generates new rows rather than having it be a part of the matrix representation in itself, as was the case with the Macaulay matrix construction. The only difference from F_4 using this representation rather than the Macaulay one (and adapting the algorithm) is that the resulting matrices only contain polynomials of a certain degree (rather than polynomials of at most that degree) and that the generation of rows for the matrix is done under the criteria that the new polynomials must be part of some critical pair.

In the paper proposing F_4, Faugére states that the algorithm analytically requires $O(2^{2^n})$ *arithmetic operations* in the worst-case generally and $O(d^{3n})$ in the worst-case for most systems, and thus does not outperform Buchberger's algorithm in the worst-case. On the other hand does Faugére present impressive experimental results that vouches for the merit of the algorithm, which is shown partly in faster solvings of previously tractable problems and partly in solving problems intractable to all previously known methods.

8.1.1 The matter of semi-regularity

Due to the unappealing time complexity for F_4 in the generic case, the question arises if there is a lower estimation to be reached under some assumptions about E. From the realisation that the performance of the F_4 algorithm is decided to a large part by the size of the involved matrices, and that these sizes depend on the degree of the involved polynomials, the fundamental role of the largest degree of a polynomial occurring in the Gröbner basis calculation becomes clear.

In [BFS03], Bardet, Faugére and Salvy define the case of *semi-regular overdetermined sequences*. The importance of overdetermined systems is, amongst other reasons, due to that overdetermined systems may sometimes be created from determined or underdetermined systems, for example by inclusion of the field equations and thus resulting in a new system with $m + n$ equations. The interest in semi-regularity is partly due to a result of a bound on the highest degree of an element in a Gröbner basis for regular systems, and partly from a conjecture that most overdetermined systems are semi-regular. Also, the idea of regularity corresponds to a notion of randomness in the polynomial system, and thus the studying of regular (or semi-regular) systems aligns well with the idea that systems arising in cryptography should ideally, from the point of view of the designer, be "random-looking".

Definition 8.2 (Semi-regular sequence, formulation from [Ars+04])**.**
Let $(P_1, ..., P_m)$ be a sequence of homogenous polynomials in $R := \mathbb{F}_2[\mathbf{x}]^m/[x_1^2 - x_1, ..., x_n^2 - x_n]$ and $\mathcal{I} := \langle P_1, ..., P_m \rangle$ an ideal of R. Then:

- *The degree of regularity of $\mathcal{I}$ is the minimial degree D_{reg} such that $\{LT(P)|P \in \mathcal{I}, deg(P) = D_{reg}\}$ is identical to the set of all monomials of degree D_{reg} in R.*
- *$(P_1, ..., P_m)$ is homogenous semi-regular on R if $R \neq \mathcal{I}$ and $\forall i \in \{1, ..., m\}$, if for any $g_i \in R$ such that $g_i P_i = 0$ in $R \setminus \langle P_1, ..., P_{i-1} \rangle$ and $deg(g_i P_i) < D_{reg}(\mathcal{I})$, then $g_i = 0$ in $R \setminus \langle P_1, ..., P_i \rangle$*

Let $(P_1, ..., P_m)$ be a sequence of affine polynomials in R and let $(P_1^h, ..., P_m^h)$ be the homogenous parts of the highest degree in each respective P_j. Then:

- *$(P_1, ..., P_m)$ is semi-regular on R iff $(P_1^h, ..., P_m^h)$ is homogenous semi-regular on R.*

- *The degree of regularity D_{reg} of $\mathcal{I} := \langle P_1, ..., P_m \rangle$ is the same as the degree of regularity of $\langle P_1^h, ..., P_m^h \rangle$.*

Another helpful intuition about (semi-) regularity is that the degree of regularity may also be seen as a measure of how independent the polynomials in the system are, more concretely, for example, as a measure of the linear independence between the rows in a simple matrix representation of E. This intuition corresponds well to the definition above, in that a low level of regularity means that the elements of the ideal already for a low maximal degree, in some sense, spans all possible monomials of that degree. This intuition also relates closely to that of a low degree of regularity relating to a system "looking random", since one would expect a system that is randomly generated to exhibit largely independent polynomials but that as the number of polynomials increase, they have to become more and more dependent and thus this line of thinking also motivates belief in the conjecture that essentially all overdetermined systems are semi-regular.

F_4 birthed the development of a few other notable algorithms, such as:

- Improved F_4: In the paper proposing the basic F_4 algorithm outlined above, the author continues to an improvement of the algorithm. This improved algorithm firstly utilises a criterion, *Buchberger's* criterion, to more carefully decide which pairs to include in *Pairs* in the beginning. Secondly, the matrices from earlier iterations are kept rather than disregarded completely, and are used to simplify new polynomials.
- F_5: In 2002, Faugére continues his work on F_4 by proposing a new variant, F_5[Fau02], where, with algebraic means, some pairs that will be reduced to 0, and thus not contribute to the Gröbner basis, are sorted out. This improvement does once again beat previous methods practically for some problems but does not improve the analytical complexity.
- `Hybrid-`F_5 [BFP09]: The idea, that also pops up in many other methods, of guessing some number k of variables before starting the main algorithm is used. The hope for this is that the guessing should allow the remaining variables to be solved with a lower D_{reg}, and thus incurr increased efficiency.

Whilst none of these methods decrease the worst-case complexity for a general system, the complexity for semi-regular systems is lowered the most, as far as we are aware, by `Hybrid-`F_5 to

$$O\left(2^k \cdot m^\omega \cdot \binom{n-k-1+D_{reg}(n-k)}{D_{reg}(n-k)}^\omega\right) [\textit{Arithmetic operations}],$$

or less generally by F_5 to

$$O\left(m^\omega \cdot \binom{n-k-1+D_{reg}}{D_{reg}}^\omega\right) [\textit{Arithmetic operations}],$$

where $2 \leq \omega \leq 3$ is a linear algebra constant stemming from the complexity reducing a matrix [BFP09]. The idea behind the derivation of this complexity is that the step dominating the computation time is *Reduce* for the largest of the matrices. This step takes about the size of the matrix to the power of ω arithmetic operations, and it is shown that the size of the largest matrix will be about $\binom{n+D_{reg}}{D_{reg}}$ elements. The hybridisation adds a 2^k term for the number of times the guessing is repeated but allows to switch D_{reg}, the degree of regularity of the original system, to $D_{reg}(n-k)$, the degree of regularity for the system after k variables have been fixed.

Whilst we have found no explicit proof that the complexity for F_5 holds also for F_4 in the case of semi-regular systems, a closer look at the complexity analysis of F_5 yields that it appears unaffected by the differences between the two algorithms (more precisely does the size of the largest involved matrix not change in the worst-case). This view that the two algorithms share complexity for semi-regular systems in the worst-case is also supported by that it appears common knowledge within the commmunity of algebraic cryptanalysis, for example does Nakamura et al. without further motivation claim that F_4 has the complexity given above [Nak+20].

8.2 FGLM, 1993

Correctness assumptions:	A Gröbner basis is known, the degree $deg(\mathcal{I})$ of the ideal is finite
Worst-case time complexity:	$O(n \cdot D(\mathcal{I})^3)$ [Arithmetic operations]
Space complexity:	*Unknown*
Note:	Outputs a reduced Gröbner basis under new ordering, not roots

From experience, and experimental evidence, it is known that the monomial ordering used is critical for the performance of a Gröbner basis algorithm, for example Buchberger's or F_4. For this purpose, it is commonplace to use grevlex (see, for example, [Fau+93] or [CGH91]), however, in order to extract the roots of the ideal one may not use grevlex but rather something like lex, and thus one is faced with two options: Either one accept the slower calculation of the original Gröbner basis under lex, or one first calculate a Gröbner basis under grevlex and then translate that basis to one under lex. The second is typically faster [Fau+93] and thus the need for an algorithm like FGLM[Fau+93] presents itself.

Suppose now we have a Gröbner basis G_1 with respect to a monomial ordering $\succeq_1$ of some zero-dimensional ideal $\mathcal{I} \subset \mathbb{F}_2[\mathbf{x}]$, i.e. an ideal that has finitely many roots, and would like to translate that into a Gröbner basis G_2 with respect to another monomial ordering $\succeq_2$. Let, as before, $\mathbf{x}^{\mathbf{u}}$ denote the monomial $x_1^{u_1} \cdot ... \cdot x_n^{u_n}$ for some fix vector $\mathbf{u} \in \mathbb{F}_2^n$ of exponents.

Definition 8.3 (Natural basis and bordering of Gröbner basis, reformulating of Definitions 2.1 and 3.1 in [Fau+93])**.**
Define the natural basis $B(G)$ for some Gröbner basis G of a zero-dimensional ideal $\mathcal{I}$ as the set of the reduced monomials in $\mathbb{F}_2[\mathbf{x}]/\mathcal{I}$ with respect to G. Call $M(G) := \{x_i\mathbf{x}^{\mathbf{b}} | \mathbf{x}^{\mathbf{b}} \in B(G), i \in \{1, ..., n\}, x_i\mathbf{x}^{\mathbf{b}} \notin B(G)\}$ the bordering of G. Let $deg(\mathcal{I})$ denote the degree of $\mathcal{I}$ and define it as the dimension of $\mathbb{F}_2[\mathbf{x}]/\mathcal{I}$.

Also define $T(G)$ as the $n \times deg(\mathcal{I}) \times deg(\mathcal{I})$ tensor whose elements t_{ijk} are defined as the jth coordinate (with respect to $B(G)$) of $\overline{x_i\mathbf{x}_k^{\mathbf{b}}}^G$, where $\mathbf{x}_k^{\mathbf{b}}$ is the kth element of $B(G)$ under the same ordering as G.

The intuition behind the definition of the bordering is (for our case of working over $\mathbb{F}_2$) that it is all monomials $\mathbf{x}^{\mathbf{u}}$ not in the natural basis such that there is at least one bit in $\mathbf{u}$ that if it is flipped from 0 to 1, then the resulting monomial *is* in the natural basis. The point of the definition of $T(G)$ is that knowing $T(G)$ and $B(G)$ allows one to find $\overline{x_i\mathbf{x}_k^{\mathbf{b}}}^G$ by

$$\overline{x_i\mathbf{x}_k^{\mathbf{b}}}^G = \sum_{j=1}^{deg(\mathcal{I})} t_{ijk}\mathbf{x}_j^{\mathbf{b}}, \tag{17}$$

for any given $i \in \{1, ..., n\}$ and $\mathbf{x}_k^{\mathbf{b}} \in B(G)$, where $B(G) = \{\mathbf{x}_1^{\mathbf{b}}, ..., \mathbf{x}_{deg(\mathcal{I})}^{\mathbf{b}}\}$.

The idea of FGLM is then, essentially, to use $B(G_1) = \{\mathbf{x}^{\mathbf{a}}_1, ..., \mathbf{x}^{\mathbf{a}}_{deg(\mathcal{I})}\}, M(G_1)$ and $T(G_1)$ to find $B(G_2) = \{\mathbf{x}^{\mathbf{b}}_1, ..., \mathbf{x}^{\mathbf{b}}_{deg(\mathcal{I})}\}$ and a matrix $C \in \mathbb{F}_2^{deg(\mathcal{I})\times deg(\mathcal{I})}$ such that any element $\mathbf{x}^{\mathbf{b}}_j$ of $B(G_2)$ can be expressed as a linear combination of elements $\mathbf{x}^{\mathbf{a}}_k$ in $B(G_1)$ according to

$$\mathbf{x}^{\mathbf{b}}_j = \sum_{k=1}^{deg(\mathcal{I})} c_{kj}\mathbf{x}^{\mathbf{a}}_k. \tag{18}$$

The FGLM algorithm proceeds roughly as follows (reformulation of the proof of Proposition 4.1 in [Fau+93]):

- Iteratively construct $B(G_2)$, $M(G_2)$, C and through them G_2. Start with

  $$B(G_2) \leftarrow \{1\}, M(G_2) \leftarrow \emptyset,$$

  since 1 cannot be in the natural basis due to the ideal being zero-dimensional but the polynomial 0 trivially always is.

- Then iteratively consider

  $$\mathbf{x}^{\mathbf{u}} \leftarrow min_{\succeq_2}\{x_i\mathbf{x}^{\mathbf{b}}_j | \mathbf{x}^{\mathbf{b}}_j \in B(G_2), i \in \{1, ..., n\}, x_i\mathbf{x}^{\mathbf{b}}_j \notin B(G_2) \cup M(G_2)\}.$$

  For such monomial $\mathbf{x}^{\mathbf{u}}$, three cases may occur:

  1. $\mathbf{x}^{\mathbf{u}} = LM(g)$ for some $g \in \mathbb{F}[\mathbf{x}]$ that should enter G_2,
  2. $\mathbf{x}^{\mathbf{u}}$ should be inserted in $B(G_2)$,
  3. $\mathbf{x}^{\mathbf{u}}$ should be inserted in $M(G_2)$ but $\mathbf{x}^{\mathbf{u}}$ is a strict multiple of $LM(g)$ for some $g \in G_2$.

- For case 3, one can easily check if it holds and if so, ignore that $\mathbf{x}^{\mathbf{u}}$. In the two other cases, we may, since by design $\mathbf{x}^{\mathbf{u}} = x_i\mathbf{x}^{\mathbf{b}}_j$, compute the linear combination of elements in $B(G_1)$ that is equal to $\mathbf{x}^{\mathbf{u}}$. This is done by utilising $T(G_1)$ and the partially complete C according to

  $$\mathbf{x}^{\mathbf{u}} = x_i\mathbf{x}^{\mathbf{b}}_j = x_i \sum_k c_{kj}\mathbf{x}^{\mathbf{a}}_k = \sum_k c_{kj}(x_i\mathbf{x}^{\mathbf{a}}_k) \tag{19}$$

  $$= \sum_k c_{kj}(\sum_h t_{ihk}\mathbf{x}^{\mathbf{a}}_h) = \sum_h (\sum_k t_{ihk}c_{kj})\mathbf{x}^{\mathbf{a}}_h \tag{20}$$

  $$= \sum_h c(\mathbf{x}^{\mathbf{u}})_h\mathbf{x}^{\mathbf{a}}_h. \tag{21}$$

  At this point, the relationship between $\mathbf{c}(\mathbf{x}^{\mathbf{u}})$ and the rows already in C decide whether one is in case 1 or 2. If $\mathbf{c}(\mathbf{x}^{\mathbf{u}})$ is not a linear combination of the rows in C, then one is in case 2 and we have found a new element of $B(G_2)$ and $\mathbf{c}(\mathbf{x}^{\mathbf{u}})$ is added as a new row in C. If on the other hand $\mathbf{c}(\mathbf{x}^{\mathbf{u}})$ *is* a linear combination of rows already in C, then the dependence between $\mathbf{c}(\mathbf{x}^{\mathbf{u}})$ and the rows gives a new element g of G_2, as outlined in the pseudocode of the algorithm below.

We will need some further notations and constructions for the pseudocode of FGLM. First of all, $ListOfNexts$ be a list of monomials $\mathbf{x}^{\mathbf{u}} = x_i\mathbf{x}^{\mathbf{b}}_j$, with two operations $.append(\mathbf{x}^{\mathbf{u}})$, that appends a monomial to the list (ignoring duplicates) and sorts $ListOfNexts$ ascendingly with respect to $\succeq_2$, and $.next$, that removes the next monomial on the list and returns it together with i and j. Also,

C is represented as a list of vectors $\mathbf{c}_i$ such that $\overline{\mathbf{x}_j^{\mathbf{b}}}^{G_1} = \sum_{k=1}^{deg(\mathcal{I})} c_{jk}\mathbf{x}_k^{\mathbf{a}}$. We also denote the list that represents $LM(G_2)$ by LM_{G_2} to make clear that it is a list and not a subprocedure. We may now in Algorithm 5 present the pseudocode of FGLM (Reformulation of Procedure 4.1 in [Fau+93]):

Algorithm 5 FGLM

Require: G_1, a Gröbner basis of zero-dimensional ideal $\mathcal{I}$, w.r.t. some monomial ordering $\succeq_1$.
Require: A new monomial ordering $\succeq_2$.
Require: A tensor $T(G_1)$ and two lists of monomials $B(G_1)$ and $M(G_1)$, all defined as above.
$C \leftarrow \emptyset$, $LM_{G_2} \leftarrow \emptyset$, $G_2 \leftarrow \emptyset$, $ListOfNexts \leftarrow \emptyset$
$\mathbf{x}^{\mathbf{u}} \leftarrow 1$
for $k \in \{1, ..., n\}$ **do**
    $ListOfNexts.append(x_k\mathbf{x}^{\mathbf{u}})$
end for
$\mathbf{x}^{\mathbf{u}}, i, j \leftarrow ListOfNexts.next$
while $\mathbf{x}^{\mathbf{u}} \neq \emptyset$ **do**
    if $\mathbf{x}^{\mathbf{u}}$ not multiple of any element in LM_{G_2} **then** ▷ Not case 3
        $\mathbf{r} \leftarrow \overline{\mathbf{x}^{\mathbf{u}}}^{G_1}$
        if $\exists \lambda_k \in \mathbb{R}\ \forall \mathbf{c}_k \in C$: $\mathbf{r} = \sum_k \lambda_k \sum_j c_{kj}\mathbf{x}_j^{\mathbf{a}}$ **then** ▷ Case 1
            $g \leftarrow \mathbf{x}^{\mathbf{u}} + \sum_k \lambda_k \sum_j c_{kj}\mathbf{x}_j^{\mathbf{b}}$
            $G_2 \leftarrow G_2 \cup \{g\}$
            $LM_{G_2} \leftarrow LM_{G_2} \cup \{\mathbf{x}^{\mathbf{u}}\}$
        else ▷ Case 2
            $\mathbf{c}_{new} \leftarrow \{\sum_{k=1}^{|C|} t_{ihk}c_{kj}\}_{h=1,...,deg(\mathcal{I})}$
            $C \leftarrow C \cup \{\mathbf{c}_{new}\}$
            for $k \in \{1, ..., n\}$ **do**
                $ListOfNexts.append(x_k\mathbf{x}^{\mathbf{u}})$
            end for
        end if
    end if
    $\mathbf{x}^{\mathbf{u}}, i, j \leftarrow ListOfNexts.next$
end while
Output: G_2, a reduced Gröbner basis of $\mathcal{I}$ under the new ordering $\succeq_2$.

8.3 XL, 2000

Correctness assumptions:	$m \geq n$, $m \geq \frac{(n-D+3)(n-D+2)}{D(D-1)\mu}$
Assumptions for complexity:	$d = 2$, $D \ll n$
Expected time complexity:	$O(\binom{n}{n/\sqrt{m\mu}}^{\omega})$, μ = proportion linearly independent elements in $\mathcal{I}_D$ [CP03]
Space complexity:	*Unknown*
Note:	Expected time complexity is polynomial if $m \geq \frac{n(n+1)}{2}$

Idea: *Generate more equations for the original system so that the resulting system can be solved efficiently by linearisation!*

In their paper from 2000 [Cou+00], Courtois et al. revisit the idea of *relinearisation*, which boils down to the goal of reformulating a non-linear equation into an equivalent system of linear equations by introducing new variables. Due to its appealing simplicity and impressive performance estimates, the XL algorithm has been followed up by an enormous number of improvements and adaptations. Three surveys of the XL family of algorithms, including their relationship to Gröbner basis algorithms, that have provided a base for this subchapter and the next are [TW10][AM11] [YC04].

Definition 8.4 (XL algorithm, reformulation of Definition 1 in [Cou+00])**.**
Choose some $D \in \mathbb{N}$. Let $E := \{P_1, ..., P_m\}$ be a set of polynomials and let $\mathcal{I}_D$ denote the set of all polynomials of degree at most D that may be generated by multiplying a polynomial in E by a monomial. Note that thus may all elements in $\mathcal{I}_D$ can be written as $P_j \mathbf{x}^\alpha$, where $\mathbf{x}^\alpha$ is some monomial of degree at most $D - deg(P_j)$. The XL algorithm then consists of performing the following steps:

1. ***Multiply:*** *Generate all the products $P_j \mathbf{x}^\alpha \in \mathcal{I}_D$.*
2. ***Linearise:*** *Consider each monomial in x_i of degree less than or equal to D as a new variable and perform Gaussian elimination on the equation system obtained in step 1. The ordering on the monomials must be such that all the terms containing some variable, say x_1 are eliminated last.*
3. ***Solve:*** *Assume that step 2 gives at least one univariate equation in the powers of x_1. Solve this equation over the finite field.*
4. ***Repeat:*** *Simplify the equations (substitute for x_1) and repeat from step 1 to find the values of the other variables.*

The performance of XL is, as becomes evident from the algorithm itself, mostly govenered by the choice of the parameter D, since it is what regulates the size of the matrices that will be reduced. The designers of XL put it similarily when they say that the idea of the algorithm " is to find some $\mathcal{I}_D$ ... which is easier to solve than the initial set of equations ..." [Cou+00]. Obviously, D should ideally be as small as possible, but it needs to be of a sufficiently large size such that the algorithm terminates successfully. Denote by D_0 the *solving degree* for XL for some specific system dimensions, the lowest choice of D such that such systems can reliably be solved. In 2004 [YC04], Yang and Chen state, refering to a proof of Claus Diem, that under some assumptions then one can calculate D_0 as

$$D_0 := min\{D | coeff((1-t)^{m-n-1} \cdot (1+t)^m, t^D) \leq D\}, \tag{22}$$

where *coeff*$(P(t), t^D)$ means the coefficient of t^D in some univariate polynomial $P(t)$.

Perhaps most notably, the authors proposing `XL` show that the algorihm has a *polynomial expected runtime for severly overdetermined quadratic systems*, more precisely: if $m \geq \epsilon \cdot n^2$, $\epsilon \in (0, \frac{1}{2}]$ then the expected runtime is about

$$2^{O(\frac{1}{\sqrt{\epsilon}})}[\textit{Unit unspecified}]. \tag{23}$$

Courtois and Patarin gives a more detailed estimate of the runtime of `XL` in 2003 [CP03] when they claim the *expected* complexity of `XL` as

$$\binom{n}{n/\sqrt{m\mu}}^{\omega}[\textit{Unit unspecified}], \tag{24}$$

where μ is the proportion of linearly independent elements in $\mathcal{I}_D$ and $2 \leq \omega \leq 3$ is the coefficient associated with performing gaussian elimination.

As touched upon above, there have been many adaptations of the `XL` algorithm, for example by guessing some variables or changing the method of matrix reduction, and some of them are described shortly below:

- `FXL` (2000) [Cou+00]: In *Fix-XL (FXL)*, the idea is similarily to in `Hybrid-`F_5 to guess some variables before starting the main algorithm and this is done with the aim of decreasing the solving degree and thus achieve a higher efficiency.
- `XL'` (2003) [CP03]: Courtois and Patarin notes that sometimes `XL` fails for a degree D, i.e. does not manage to find a univariate polynomial, only for the reason of having generated slightly too few equations. Therefore they devise a revised version of `XL` that instead of looking for a single univariate polynomial looks for a subsystem of some small number of r equations in only r variables. This small subsystems is then solved by exhaustive search.
- `XL2` (2003) [CP03]: Courtois and Patarin continues with the design of `XL2`, where the idea is to let the linearisation lead to one polynomial in a restricted number of monomials and then generate new polynomials from it in a way such that these polynomials were not in the reduced matrix, thus allowing E to be solved at a lower D.
- MXL (2008) [Din+08]: Similarily to in `XL2`, in *mutant-XL (MXL)* the matrix is extended if there are not enough equations to find a univariate polynomial. Jintai Ding however notes critically that these new polynomials can be created such that their degrees are deliberately kept as small as possible.
- `WXL` (2011) [Moh+10]: Mohammed et al. notes that since the matrices occurring in the `XL` algorithm tend to be sparse, which may be realised if, for example, one assumes E is generated uniformly at random and has $d \ll n$. Mohammed et al. proposes therefore to employ the *block Wiedemann algorithm* [Wie86] for matrix reduction instead of gaussian elimination, since it is specifically geared towards sparse matrices.
- PXL (2021) [FK21]: Recently Furue and Kudo proposed the *polynomial-XL (PXL)* algorithm with the idea to improve `FXL` by only *partly* reducing the matrix before fixing some variables and

then finishing the reduction. An increase in efficiency is supposedly gained for some parameter choices due to that the work needed to be done for each fixing can be decreased.

8.4 The relationship between Gröbner basis algorithms and XL

The overarching approach in both XL and F_4 is the same, namely to extend the set E of polynomials into a matrix and then reduce that matrix. The complexities of the algorithms also show some common traits, namely that they are dominated by reducing the largest matrix involved and that the algorithms thrive from having $m > n$ and a low dependence between the polynomials. The differences between the two algorithm such as that XL searches for a solution of E whilst F_4 looks for a Gröbner basis are also not as large as one might initially think. In 2004, Ars et al. [Ars+04] shows that the original XL algorithm is to be seen as a redundant version of F_4, in the sense of that:

1. The problem of solving PoSSo is entirely equivalent to finding a Gröbner basis, under the assumption that the solution is unique.
2. Thus XL is actually a Gröbner basis algorithm in this case, and the fact that it does not introduce an explicit monomial ordering may be disregarded due to a result that if the algorithm terminates for some input, then it will also terminate for the same input under a monomial ordering.
3. The steps of XL can be seen as equivalent to the steps in F_4, but with the redundancy added that XL will generate larger matrices than F_4.

In order to prove their results, Ars et al. give a formulation of XL that is F_4-like but equivalent in practice to the one given in the original XL paper. In Algorithm 6 below we give a reformulation of the algorithm by Ars et al. and it is to be compared to the definition of the XL algorithm given in the previous subchapter. A major change that Ars et al. does to the original formulation of XL is that it is argued that in order to find a suitable D, all reasonable implementations involve iteratively increasing D in some manner during the course of the algorithm.

Algorithm 6 XL (F_4-like)

Require: A set $E := \{P_1, ..., P_m\}$ of polynomials.
$G \leftarrow E$, $k \leftarrow 0$
$Pairs \leftarrow \{Pair(P_i, P_j) | P_i, P_j \in E : i \neq j\}$
while $Pairs \neq \emptyset$ **do**
    $k \leftarrow k + 1$
    $M, Pairs_k \leftarrow GenMatrix_{XL}(Pairs, k)$
    $Pairs \leftarrow Pairs \setminus Pairs_k$
    $\tilde{M} \leftarrow Reduce(M, G)$
    for $h \in \tilde{M}$ **do**
        $Pairs \leftarrow Pairs \cup \{Pair(h, g) | g \in G\}$
        $G \leftarrow G \cup h$
    end for
end while
Output: Gröbner basis G of $\mathcal{I} := \langle P_1, ..., P_m \rangle$

Algorithm 7 $GenMatrix_{XL}$

Require: A set $Pairs$ of critical pairs. An integer k.

1: $Pairs_{used} \leftarrow \{p \in Pairs | deg(p) \leq k\}$
2: $M \leftarrow \{tg | (t, g) \in Left(Pairs_{used}) \cup Right(Pairs_{used})\}$
3: **Return:** $M, Pairs_{used}$

Algorithm 8 *Reduce*

Require: A set M of polynomials.

1: $\tilde{M} \leftarrow RowEchelonForm_{\succeq}(M)$
2: $\tilde{M} \leftarrow \{f \in \tilde{M} | LT(f) \notin LT(M)\}$
3: **Return:** $\tilde{M}$

Comparing Algorithm 6 to the formulation of F_4 above in Algorithm 2, it is easy to see that the only difference is the way in which the matrix M is generated in each iteration. Relating to the earlier discussion about the matrix representation in F_4, we can note that `XL` does build Macaulay matrices of the polynomials in $Pairs$ and thus this difference in matrix generation may also be seen, in essence, simply as a difference in choice of matrix representation. A main point made in [Ars+04] is that the approach used in $GenMatrix_{XL}$ is worse than the one in $GenMatrix_{F_4}$ because it generates larger matrices.

This result perhaps disregarded, the development of new flavors of `XL` continued after 2004, as shown in the previous subchapter. This development, perhaps most notably in the introduction of MXL[Din+08], leads in 2011 to another work [Alb+12] relating this new `XL` version and F_4. In this paper, Albrecht et al. conclude similarily to the work presented in 2004 that this new branch to the family of `XL` algorithms also should be regarded as redundant versions of F_4. Neither did this result stop the interest in new variants of `XL`, for example did Cheng et al. present an improved parallelised implementation of `WXL` in 2016 [Che+16] and Furue and Kudo introduced PXL, in 2021 [FK21].

The continued research efforts regarding improving `XL` might be seen as that perhaps the relationship between that family and that of the Gröbner basis algorithms is not fully understood and agreed upon. For example does Cheng et al. claim that their `WXL` version is much more memory efficient than the implementation of F_4 in MAGMA, an implementation that is "often used as a standard benchmark for cryptographers" [Che+16]. It is although unclear whether improvements such as these are due to some inherent difference between the `XL` and Gröbner basis algorithm families or if the improvements made to `XL` may also be made to F_4 with similar consequences. It should however be noted that, at the time of writing, we are not aware of any direct critisism of the conclusions drawn in [Ars+04] and [Alb+12].

9 Algorithms using various known techniques

As we noticed in Chapter 7 and Chapter 8, there are some high-level design ideas that are present in several of the algorithms, such as fixing a small number of variables or reducing a linear system. For those algorithms, these common traits are perhaps mostly seen as tweaks to core algorithms that themselves are fundamentally different. For the methods in this chapter, however, we take another angle on it and say that these algorithms are to be seen as having the primary trait that they are mixes between ideas present in previous algorithms. This view on the approaches in this chapter has been a part of the discourse for a few years, perhaps most explicitly in the paper proposing a simple deterministic algorithm[BDT21], which we will discuss in the first subchapter.

9.1 A simple deterministic algorithm (BDT), 2021

Correctness assumptions:	$d = 2$
Assumptions for complexity:	Heuristic randomness assumptions (see below)
Expected time complexity:	$O(n^{\frac{3}{2}}2^{n-\sqrt{2m}})$ [Arithmetic operations]
Space complexity:	*Negligible*

Idea: *Summarise the common ideas of fixing variables and linearising the system in an algorithm as simple to understand and implement as possible!*

In 2021, Bouillaguet, Delaplace and Trimoska introduce an algorithm for solving $\mathcal{MQ}$, that they call *a simple deterministic algorithm* [BDT21] but that we will henceforth call BDT due to the inconvenient length of the original name. The authors also present at detail a point of view on a part of the landscape of previously available algorithms. They argue that some of the algorithms in the state-of-the-art (most directly BooleanSolve and Crossbred but also XL and F_4/F_5) can be seen as generalisations of their new algorithm and as opposed to almost all other papers in this survey they claim a *decremental* improvement over the state-of-the-art. With this they mean that whilst BDT is asymptotically much slower analytically than other approaches, BDT is still to be seen as an improvement since it is efficiently implementable and extremely simple to understand.

The whole BDT algorithm may in fact be explained as [BDT21]:

"Guess sufficiently many variables so that the remaining polynomial system can be solved by linearisation, i.e. by considering each remaining monomial as an independent variable, solving the resulting linear system and checking each solution against the original system".

In more detail this means:

1. Choose a parameter $k := n - \lfloor\sqrt{2m} - 2\rfloor$. Partition $\mathbf{x}$ into $(\mathbf{y}, \mathbf{z})$ by $\mathbf{y} := (x_1, ..., x_k), \mathbf{z} := (x_{k+1}, ..., x_n)$.
2. Compute the set $\mathcal{L}$ of linear combinations of $\{P_1, ..., P_m\}$ such that they become linear in $\mathbf{z}$ after $\mathbf{y}$ has been fixed, i.e. $\mathcal{L} := \{P \in \langle P_1, ..., P_m\rangle | deg(P(\mathbf{y}^*, \mathbf{z})) \leq 1, \mathbf{y}^* = (1, ..., 1)\}$.

3. For each fixing of $\mathbf{y}$, solve the resulting linear system and test the proposed solution on E.

This explaination leads, in pseudocode, to the algorithm below (reformulation of Algorithm 1 in [BDT21]):

Algorithm 9 `BDT`

Require: A quadratic polynomial system E. Two integers $l \in \{1, ..., m\}$ and $k \in \{0, ..., n\}$.
1: Compute $\mathcal{L}$, the set of linear combinations of P_j such that each monomial contains at most one z_i.
2: Compute a basis $\{g_1, ..., g_l\}$ of $\mathcal{L}$
3: Factorise $g_i(\mathbf{y}, \mathbf{z}) = q_i(\mathbf{y}) + \mathbf{y}B_i\mathbf{z}^t + C_i\mathbf{z}^t$ for all $i \in \{1, ..., l\}$.
4: **for** $\mathbf{y}^* \in \mathbb{F}_2^k$ **do**
5: $M, \mathbf{b} \leftarrow GenMatrix_{BDT}(\{q_i\}, \{B_i\}, \{C_i\}, \{\mathbf{y}^*\})$
6: $\tilde{M}, \tilde{\mathbf{b}} \leftarrow Reduce_{BDT}(M, \mathbf{b})$
7: $Sols \leftarrow Solve(\tilde{M}, \tilde{\mathbf{b}})$
8: **for** each solution $\mathbf{z}^* \in Sols$ **do**
9: **if** $E(\mathbf{y}^*, \mathbf{z}^*) = 0$ **then**
10: **Return** $(\mathbf{y}^*, \mathbf{z}^*)$
11: **end if**
12: **end for**
13: **end for**
14: **Return:** $\emptyset$

Algorithm 10 $GenMatrix_{BDT}$

Require: l polynomials $q_i(\mathbf{y})$, l matrices $B_i \in \mathbb{F}_2^{k \times (n-k)}$
Require: l vectors $C_i \in \mathbb{F}_2^{(n-k)}$, A vector $\mathbf{y}^* \in \mathbb{F}_2^k$
1: **for** $i \in \{1, ..., l\}$ **do**
2: $b_i \leftarrow q_i(\mathbf{y}^*)$
3: $M_{i\cdot} \leftarrow \mathbf{y}^* B_i \oplus C_i$
4: **end for**
5: **Return:** $M, \mathbf{b}$

The $Reduce_{BDT}$ consists of only Gaussian elimination of $M\mathbf{z} = \mathbf{b}$, as given by $GenMatrix_{BDT}$, and finding the solutions from the reduced matrix equation is done directly through back substitution.

In [BDT21], the authors argue that the runtime is dominated by generating the matrices. This involves evaluating l quadratic polynomials in k variables, taking $O(lk^2)$ operations, and performing l matrix-vector products of the size $k \times n - k$, taking $O(lk(n-k))$. The only other step in `BDT` that takes a substantial amount of time is the solving of the linear systems (step 6 and 7 in `BDT`), which takes in total $O(l(n-k)^2)$ operations, dominated by the Gaussian elimination. This follows from an argument that under the *heuristic assumption* that $\mathbf{b}$ is uniformly random, that the step of testing proposed solutions is asymptotically neglible compared to the other steps of the algorithm.

The runtime of generating (and maintaining) the matrices can be made more efficient by utilising the folklore differential technique, as in `FES`, and rearranging the algorithm. The runtime of evaluating the quadratic polynomials then becomes $O(k)$ operations per point in the input space and polynomial, and the runtime for updating the matrix likewise become $O(n-k)$ operations. Therefore the complexity of generating and maintaining the matrices now match the complexity of solving the linear systems

and the total runtime of BDT becomes

$$2^k(O(lk) + O(l(n-k)) + O(l(n-k)^2)). \tag{25}$$

The final step of the runtime analysis is then to decide on parameters l and k. The designers of the algorithm choose to set $n-k := \lfloor \sqrt{2m} - 2 \rfloor$, guaranteeing there is less than m non-constant quadratic monomials in $\mathbf{z}$ and thus allowing the subsystem to be solved by linearisation. They also choose $l := m - \frac{(n-k)(n-k-1)}{2}$. These choices result in, under the assumption than $m \approx n$, that $k = O(n)$, $n-k = O(\sqrt{n})$ and $l = O(\sqrt{n})$. Thus the total runtime of BDT is

$$2^{n-\lfloor\sqrt{2m}-2\rfloor}(O(n^{\frac{3}{2}}) + O(n) + O(n^{\frac{3}{2}})) = O(n^{\frac{3}{2}}2^{n-\sqrt{2m}}). \tag{26}$$

We emphasise, BDT only deals with quadratic polynomial systems. The authors of BDT note that if one extends their algorithm to systems of higher degree then two reasonable such extension would lead to the FXL, BooleanSolve and Crossbred algorithms. Firstly, one can note that BDT is quite similar to FXL except for the fact that there is no extension of the initial system and if one chooses to extend the matrix (by generating a Macaulay matrix) then one arrives at an algorithm that is quite close to FXL. The reason that extension is needed is partly because it relaxes slightly the demands of overdeterminedness and partly because it allows working with polynomials of higher degrees.

9.2 BooleanSolve, 2013

Correctness assumptions:	$m \geq n$, $d = 2$
Assumptions for complexity:	Heuristic assumptions regarding semi-regularity
Expected time complexity:	$O(2^{0.792n})$ [Arithmetic operations]
Space complexity:	*Exponential* [BDT21]
Note:	Solves the exhaustive version of $\mathcal{MQ}$

Idea: *Instead of building the Macaulay matrix and then fixing variables as in FXL, first fix variables and then build the Macaulay matrix!*

In 2013, Bardet, Faugére, Salvy and Spaenlehauer present their BooleanSolve algorithm[Bar+13] and, as Bouillaguet et al. note [BDT21], it mainly differs from previous algorithms such as FXL in that the order of fixing and matrix generation is switched. This change is done partly since it results in a slightly faster algorithm asymptotically (the most optimal FXL analysis for $\mathcal{MQ}$ at the time was supposedly $O(2^{0.875n})$ [YC04] and BooleanSolve needs in expectation $O(2^{0.792n})$ time) and partly because the assumptions needed to reach such an estimate are supposedly significantly more convenient to motivate. The authors of BooleanSolve also contribute a thorough discussion on the optimal fraction of variables to fix, a discussion also of interest for many other algorithms.

Through somewhat technical heuristic arguments and assumptions, the *expected* time complexity of BooleanSolve is shown to be $O(2^{0.792n})$ *field operations*. Whilst this is asymptotically very impressive, one should note that there seems to be large challenges with actually implementing the algorithm, as Bouillaguet et al. write in [BDT21]: "the algorithm is likely hard to implement because it requires a sparse linear system solver for exponentially large matrices. To the best of our knowledge, no implementation has ever been written."

9.3 `Crossbred`, 2018

Correctness assumptions:	*None*
Assumptions for complexity:	Semi-regularity
Expected time complexity:	$\tilde{O}((\sum_{d_k=D_{kMac}+1}^{D} \sum_{d'=0}^{D-d_k} \binom{k}{d_k}\binom{n-k}{d'})^2)$ $+2^{n-k}\tilde{O}((\sum_{i=0}^{D_{kMac}} \binom{k}{i})^{\omega})$ [Arithmetic operations][Dua20]
Space complexity:	*Unknown*

Idea: *Reducing the Macaulay matrix such that the resulting system is not linear after fixing variables but of some degree $D_{kMac} \geq 1$ is faster than a full reduction.*

The `Crossbred` algorithm[JV18] by Joux and Vitse in 2018 may be seen as an extension of `BDT` into a case that allows for $d > 2$. The main extending idea there is that instead of operating directly on the polynomials in E, the Macaulay matrix of E for some degree D_{mac} is calculated first and then the rows of this matrix are fed into a `BDT`-like procedure. As will be evident from the algorithm below, when $d = 2$ (and no extension of `BDT` therefore is needed), then `Crossbred` and `BDT` are identical. When $d > 2$, then the step in `BDT` that computes linear combinations of $\{P_1, ..., P_m\}$ such that they are linear after the variable fixing needs a bit more care. As noted above, here the designers of `Crossbred` opts to not necessarily only consider linear combinations that become linear but rather those with a degree at most $D_{kMac} \geq 1$. This is done with the reasoning that it might be worth to perform `FES` on these non-linear polynomials, rather than simple back substitution on the reduced linear ones, in order to avoid the process of seeking those polynomials linear after variable fixing.

Below follows a pseudocode for `Crossbred`, adapted primarily from the one given in [NNY18]:

Algorithm 11 `Crossbred`

Require: A polynomial system E. Integers $D, D_{kMac}, k_{outer}, k_{inner}$

for $\mathbf{z}^* \in \mathbb{F}_2^{k_{outer}}$ **do**
  $E_{\mathbf{z}^*}(\mathbf{y}) \leftarrow E(\mathbf{y}, \mathbf{z}^*)$
  $M \leftarrow GenMatrix_{Crossbred}(E_{\mathbf{z}^*}, D)$
  $\tilde{M} \leftarrow Reduce_{Crossbred}(M, D_{kMac})$
  Call $FastEvaluate(\tilde{M}, k_{inner}, n - k_{inner})$ and
  for each returned linear system $\tilde{M}'$ **do**
    if $\tilde{M}'$ is solvable **then**
      $Sols \leftarrow ExhaustiveSearch(\tilde{M}')$
      for each solution $\mathbf{y}^* \in Sols$ **do**
        if $E(\mathbf{y}^*, \mathbf{z}^*) = 0$ **then**
          Return $(\mathbf{y}^*, \mathbf{z}^*)$
        end if
      end for
    else
      Continue with $FastEvaluate$
    end if
  end for
end for
Return: $\emptyset$

Algorithm 12 $GenMatrix_{Crossbred}$

Require: A set E' of m polynomials in $n - k_{outer}$ variables. Positive integers D_0, k_{inner}.

Introduce a monomial ordering $\succeq$ on the monomials in E' as grevlex on the degrees in the first k_{inner} variables.
$M \leftarrow MacaulayMatrix_{D_0,\succeq}(E')$
Return: M

Algorithm 13 $Reduce_{Crossbred}$

Require: A set M of polynomials. A positive integer D_{kMac}.

$\tilde{M} \leftarrow$ a system of all the linearly independent polynomials in M that have degree at most D_{kMac} in the first k_{inner} variables.
Return: $\tilde{M}$

Algorithm 14 *FastEvaluate*

Require: A polynomial system $\tilde{M} := \{m_1, ..., m_{|\tilde{M}|}\}$ in variables $\{y_1, ..., y_{n-k_{outer}}\}$. Positive integers l, k_{inner}.

Require: A function $CallBack(\tilde{M}')$ that pauses execution of this current procedure and proposes $\tilde{M}'$ to the main loop to be solved.

1: **if** $l = k_{inner}$ **then**
2: $\quad Callback(\tilde{M})$
3: **else**
4: $\quad$ Factor $m_i = m_i^{(0)} + y_l m_i^{(1)}$
5: $\quad y_l \leftarrow 0$
6: $\quad FastEvaluate(\{m_1^{(0)}, ..., m_{n-k_{outer}}^{(0)}\}, l-1, k_{inner}, CallBack)$
7: $\quad y_l \leftarrow 1$
8: $\quad FastEvaluate(\{m_1^{(0)} + m_1^{(1)}, ..., m_{n-k_{outer}}^{(0)} + m_{n-k_{outer}}^{(1)}\}, l-1, k_{inner}, CallBack)$
9: **end if**

Whilst `Crossbred` looks a lot like `FXL`, `BooleanSolve` or `BDT`, there are a few key improvements to be remarked upon. Firstly one can consider the more intricate way of generating and reducing the matrix, such as prescribing the grevlex ordering, and note that this is done in order to facilitate the extraction of polynomials with a low degree in the variables remaining after fixing, as described in the previous paragraph. Secondly, one should note that the recursive fast evaluation as well as the testing of system consistency before performing exhaustive search is done and this with the hope that this is more efficient than simply solving the subsystems directly.

The performance of `Crossbred` was demonstrated experimentally in the paper [JV18] proposing the algorithm as well as in a 2018 paper [NNY18] by Niederhagen, Ning and Yang about implementing the algorithm efficiently. Especially the latter of the papers go into quite some detail about optimal choices of the parameters D, k_{inner}, k_{outer} and discusses the algorithm's behaviour for some different types of $\mathcal{MQ}_2$-Search problem instances.

In 2020 João Duarte published a pre-print [Dua20] giving an analytical estimate of the time complexity. Duarte slightly reformulates the algorithm to better suit the case of allowing degrees larger than 2 in the original system. The primary way this is achieved is by the introduction of a new parameter D_{kMac}, which is the demanded maximum degree of $\tilde{M}$, a value that was previously fixed at 1. He also removes the outer hybridisation. For his slightly changed version of `Crossbred`, Duarte estimates the time complexity as

$$\tilde{O}\left(\left(\sum_{d_k=D_{kMac}+1}^{D}\sum_{d'=0}^{D-d_k}\binom{n-k_{inner}}{d_k}\binom{k_{inner}}{d'}\right)^2\right) + 2^{k_{inner}}\tilde{O}\left(\left(\sum_{i=0}^{D_{kMac}}\binom{n-k_{inner}}{i}\right)^{\omega}\right) \quad (27)$$

arithmetic operations.

Duarte also compares his result to complexity estimates for `FES` and Hybrid-F_5 for some specific systems and one key take away is that `Crossbred` seems to be essentially exactly as efficient as `Hybrid-`F_5 in theory for these systems and that there therefore is much research left to be done on more thoroughly investigating the performance differences.

In Figure 4 below is an illustration of `Crossbred`'s behaviour for one example execution. The illus-

tration shows the entire input domain, $\mathbb{F}_2^n$, and some set of roots that we denote by S. Then there is three iterations of the main loop in the algorithm, each generating a set candidate solutions, denoted by $pSols_i$ (if i is the iteration number), that are all tested on the original system E. In the example, the algorithm ends after the third iteration since then at least one of the roots of E is proposed as a candidate solution.

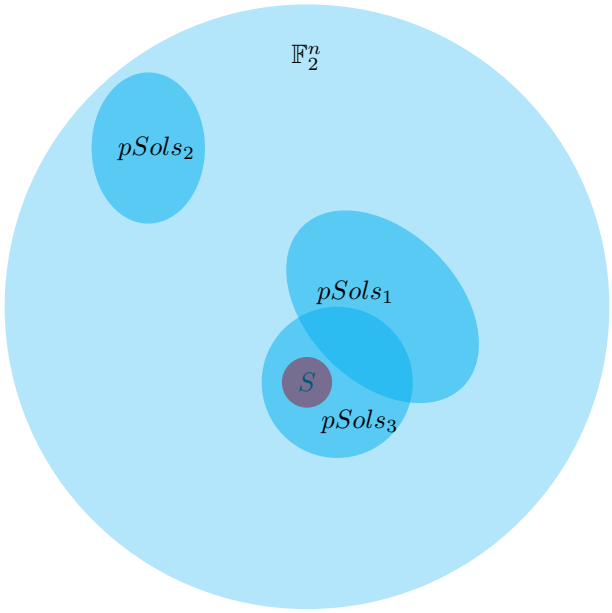

Figure 4: Illustration of the input space, solution set and candidate solution sets for an example execution of `Crossbred`.

10 The polynomial method

In recent years, a family of algorithms for solving PoSSo_2 has developed that is quite different from the approaches that we have surveyed so far. The new approach relies on what is called *the polynomial method*, which for us, roughly, means representing the polynomial system E by a single *identifying polynomial*. For another survey into these algorithms, and some first implementations, see the 2021 survey by Barbero et al. [Bar+21].

As probabilistic representations of functions will become important in this chapter, let us remark that we choose to consistently let $\tilde{f}$ denote the probabilistic representation of some function f in the sense that these evaluate equivalently for the same input with some known high probability. These probabilistic representations will however be individually defined along the way.

10.1 Williams, 2014

Idea: *The polynomial method for representing circuits as a probabilistic polynomial can also be used in algorithm design!*

The use of the polynomial method in cryptanalysis was instigated by a survey[Wil14] by Ryan Williams in 2014 in which he review some ways that the method, that was developed for use in circuit design and analysis, have been and may be used in algorithm design. The main idea of the polynomial method was originally that a circuit may faithfully be represented by a probabilistic "low-complexity" polynomial with high probability[Wil14]. The extension then to our use-case of solving polynomial systems would, roughly, be that the polynomials in the system E may be represented, with high probability, by a single probabilistic polynomial instead and that this single polynomial will have, compared to its non-probabilistic analog, a very low degree.

The mathematical construct of representing a set of polynomials by a single *identifying polynomial* is credited to Razborov and Smolensky in the 80's [Raz87][Smo87]. Let us define the *identifying polynomial* $F : \mathbb{F}_2^n \to \mathbb{F}_2$ of E as

$$F(\mathbf{x}) := \prod_{j=1}^{m} (P_j(\mathbf{x}) + 1). \tag{28}$$

This function has the property that $F(\mathbf{x}^*) = 1$ iff $\mathbf{x}^*$ is a root of E. This polynomial F, however, also has a possible maximum degree of dm and is thus in practice unwieldy to use directly. Introduce instead a variable $l \in \{1, ..., m\}$, generate $\alpha_{ij} \in \mathbb{F}_2$ uniformly random $\forall i \in \{1, ..., l\}, j \in \{1, ..., m\}$ and then define

$$R_i(\mathbf{x}) := \sum_{j=1}^{m} \alpha_{ij} P_j(\mathbf{x}), \tag{29}$$

$$\tilde{F}(\mathbf{x}) := \prod_{i=1}^{l} (R_i(\mathbf{x}) + 1). \tag{30}$$

Then $\tilde{F}$ is a probabilistic representation of F in the sense that

$$\mathbb{P}[\tilde{F}(\mathbf{x}^*) = 1 | F(\mathbf{x}^*) = 1] = 1, \tag{31}$$

$$\mathbb{P}[\tilde{F}(\mathbf{x}^*) = 0 | F(\mathbf{x}^*) = 0] \geq 1 - \frac{1}{2^l}, \tag{32}$$

for all $\mathbf{x}^* \in \mathbb{F}_2^n$ [Din][Smo87].

10.2 Lokshtanov et al., 2017

Assumptions:	*None*
Worst-case time complexity:	$O^*(2^{(1-\frac{1}{5d})n})$ [RAM model]
Space complexity:	$O^*(2^{(1-\frac{1}{5d})n})$ [Din21]
Note:	Solves PoSSo$_2$-Decision

Idea: *By solving the decision version of the problem instead of the search version the probabilistic construction becomes easier!*

In 2017, Lokshtanov et al. present the first usage of the polynomial method in the solving of polynomial systems [Lok+]. Not only does it introduce a new approach to the problem but it also is the first algorithm that achieves an exponential speedup over exhaustive search in the worst case setting without any major algebraic assumptions. Whilst other previous approaches have achieved exponential speedup over exhaustive search in other cases such as for overdetermined or underdetermined systems, or in the average-case under assumptions of some distribution, this algorithm requires no such restrictions.

Lokshtanov et al. solve the decision version of PoSSo and show it can be used to solve the search version in at most $2n$ repetitions. The algorithm solves PoSSo$_2$-Decision by utilising the constructions mentioned in the previous subchapter, as well as also introducing an additional one. First the authors split the variables $\mathbf{x}$ into $\mathbf{y} := \{x_1, ..., x_{n-k}\}$ and $\mathbf{z} := \{x_{n-k+1}, ..., x_n\}$ for some chosen $k := \lfloor \delta n \rfloor$, $\delta(d) \in (0, 1)$. Then they define

$$V(\mathbf{y}) := 1 - \prod_{\mathbf{z}^* \in \mathbb{F}_2^k} (1 - F(\mathbf{y}, \mathbf{z}^*)), \tag{33}$$

$$\tilde{V}(\mathbf{y}) := \sum_{\mathbf{z}^* \in \mathbb{F}_2^k} \beta_{\mathbf{z}^*} \tilde{F}(\mathbf{y}, \mathbf{z}^*), \tag{34}$$

where $\beta_{\mathbf{z}^*}$ are chosen in $\mathbb{F}_2$ uniformly random and $l := k + 2$ is used in the construction of $\tilde{F}$. The desired properties of V and $\tilde{V}$ are, for all $\mathbf{y}^*$,

$$V(\mathbf{y}^*) = 0 \iff F(\mathbf{y}^*, \mathbf{z}^*) = 0, \ \forall \mathbf{z}^* \in \mathbb{F}_2^k, \tag{35}$$

$$\mathbb{P}[\tilde{V}(\mathbf{y}^*) = 1 | V(\mathbf{y}^*) = 1] \geq \frac{1}{2}, \tag{36}$$

$$\mathbb{P}[\tilde{V}(\mathbf{y}^*) = 1 | V(\mathbf{y}^*) = 0] \leq \frac{1}{4}. \tag{37}$$

This means that if $\tilde{V}(\mathbf{y}^*) = 0$ then it is "unlikely" that some $(\mathbf{y}^*, \mathbf{z}^*)$ is a root but if $\tilde{V}(\mathbf{y}^*) = 1$ then it is likely that some $(\mathbf{y}^*, \mathbf{z}^*)$ is a root. Lokshtanov et al. notes that each individual $\tilde{V}(\mathbf{y}^*)$ can be evaluated efficiently from its low-degree coefficients using the Möbius transform, and that this is not the case for V, due to its high degree, is precisely the reason for introducing $\tilde{V}$ rather than working with V directly.

The main loop of the algorithm, which is repeated $t := 100n$ times, consists of generating a new $\tilde{V}$ and evaluating it for each $\mathbf{y}^* \in \mathbb{F}_2^{n-k}$, storing the result in a truthtable. Between the loops, one keeps track of the number of times each $\mathbf{y}^*$ gives $\tilde{V}(\mathbf{y}^*) = 1$ in a scoreboard and it is returned in the end that there exists a root of E if there is any $\mathbf{y}^*$ such that it has resulted in $\tilde{V}(\mathbf{y}^*) = 1$ at least $\frac{2t}{5}$ times.

The correctness proof of the algorithm stems from the result above that, probabilistically, $\tilde{V}$ is an identifier of V, which in turn is an identifier of F, which finally is an identifier of E. The choices of t and the fraction $\frac{2t}{5}$ stems from an analysis that guarantees a high enough success probability for both possible outcomes. More precisely is it shown that the probability of an incorrect output (of both kinds) is less than $\frac{100n}{2^n}$, which obviously becomes negligible for somewhat large n (it is less than 1 percent for $n = 18$ and less than one per one million for $n = 32$).

In Algorithm 15 below follows pseudocode for Lokshtanov et al.'s algorithm. The pseudocode is mostly a reformulation from the one in [Bar+21]. Note that an output of 1 means that there is a root and 0 means there is none.

Algorithm 15 Lokshtanov et al.'s algorithm

Require: A set $E := \{P_1, ..., P_m\}$ of polynomials. A positive integer k.

$l \leftarrow k + 2$
$Scoreboard \leftarrow (0, ..., 0)$ ▷ an array of $n - k$ zeros
$t \leftarrow 100n$
for $iter \in \{1, ..., t\}$ **do**
    $\tilde{F} \leftarrow GenerateFTilde(E, l)$
    $\tilde{V} \leftarrow GenerateVTilde(\tilde{F})$
    $TruthtableTilde{V} \leftarrow Evaluate(\tilde{V})$
    for $\mathbf{y}^* \in \mathbb{F}_2^{n-k}$ **do**
        if $Truthtable(\mathbf{y}^*) = 1$ **then**
            $Scoreboard(\mathbf{y}^*) \leftarrow Scoreboard(\mathbf{y}^*) + 1$
        end if
    end for
end for
for $\mathbf{y}^* \in \mathbb{F}_2^{n-k}$ **do**
    if $Scoreboard(\mathbf{y}^*) \geq \frac{2t}{5}$ **then**
        Return 1
    end if
end for
Return 0

Below follows the theorem analysing the time complexity of the algorithm and a sketch of the proof. The computational model used is a `RAM` (with unspecified instruction set). One should, however, note that since polynomial factors are disregarded in the results, the specific choice of instruction set and other details of the model can reasonably be kept implicit.

Theorem 10.1 (Runtime of Algorithm 15, reformulation of Theorem 1.1 in [Lok+])**.**
Given an instance of PoSSo$_2$-Decision, the probabilistic Algorithm 15 solves the problem correctly with high probability and terminates in time upper bounded by $O^*(2^{(1-\frac{1}{5d})n})$.

Proof-sketch, simplification of proof in [Lok+].
The runtime of the algorithm is bounded by t times the time needed to execute the main loop once.

- Applying a naive algorithm for *GenerateFTilde* and *GenerateVTilde* takes together $O^*(2^k)$ polynomial additions or multiplications and each such operation takes at most $O^*(M(n-k, d\cdot(k+2), 2)\cdot n^{2d})$ time, where $M(n-k, d\cdot(k+2), 2)$ is defined as the number of unique monomials in $\mathbb{F}_2^{n-k}$ of degree at most $d\cdot(k+2)$.
- By Lemma 2.2 in [Lok+], there is an algorithm performing $Evaluate(\tilde{V})$ in $O^*(2^{n-k})$ time.
- Updating the scoreboard takes $O^*(2^{n-k})$ time.

Thus the running time of one iteration of the main loop is at most

$$O^*(2^{n-k} + 2^k \cdot M(n-k, d\cdot(k+2), 2)\cdot n^{2d}). \tag{38}$$

Let k be expressed as a fraction of n, that is, say $k := \lfloor \delta(d) n \rfloor$ for some $\delta \in (0,1)$ chosen with respect to d and n. Let also δ' be the defined by $\delta' := \frac{\lfloor \delta n \rfloor}{n}$, such that $k = \delta' n$. Then the runtime can be written as

$$O^*(2^{(1-\delta')n} + 2^{\delta' n} \cdot M((1-\delta')n, d\cdot(\delta' n+2), 2)\cdot n^{2d}). \tag{39}$$

By the relation[3] than $M(n, r+1, 2) \leq n \cdot M(n, r, 2)$ for any integer $r > 0$, and since $\delta' n + 1 > \delta n$, $\delta' n \leq \delta n$, we learn that

$$M((1-\delta')n, d\cdot(\delta' n+2), 2)\cdot n^{2d} \leq M((1-\delta')n, d\cdot(\delta' n), 2)\cdot n^{2d}\cdot n^{2d} \leq M((1-\delta')n, d\cdot(\delta n), 2)\cdot n^{3d}\cdot n^{2d}.$$

Thus we can upper bound the runtime by

$$O^*(2^{(1-\delta')n} + 2^{\delta n} \cdot M((1-\delta')n, d\delta n, 2)\cdot n^{5d}). \tag{40}$$

Further we may WLOG ignore the difference between δ and δ' since they will coincide for large enough n. The goal is now, similarily to previously with complexity estimates with multiple large terms, to choose δ such that the terms are balanced and this means to choose δ such that $M((1-\delta)n, d\delta n, 2) = O^*(2^{(1-2\delta)n})$. This would mean that the running time of one iteration may be bounded by $O^*(2^{(1-\delta)n} n^{5d}) = O^*(2^{(1-\delta)n})$. Lemma 2.1 in [Lok+] gives us that $M(a, b, 2) \leq \binom{a}{b}$ for any positive integers a, b and thus the quest is, for given d, to find a δ such that

$$\binom{n-\delta n}{d\delta n} \leq O^*(2^{(1-2\delta)n}).$$

For $d = 2$, one can set $\delta = 0.1235$ and thus arrive at a total runtime for the algorithm of

$$O^*(2^{0.8765n}).$$

Similarily one may chose $\delta = \frac{1}{5d}$ for $d > 2$ and arrive at

$$O^*(2^{(1-\frac{1}{5d})n}).$$

□

Assumptions:	*None*
Worst-case time complexity:	$O^*(2^{(1-\frac{1}{2.7d})n})$ [*Unspecified model*]
Space complexity:	$O^*(2^{(1-\frac{1}{2.7d})n})$ [Din21]
Note:	Solves PoSSo$_2$-ParityCount

10.3 Björklund et al., 2019

Idea: *One can isolate solutions for some given variable partition and thus reduce the decision problem into finding the parity of the number of solutions!*

In 2019, Björklund et al. improve upon the work of Lokshtanov et al. through lowering the time complexity further [BKW19]. This is primarily done by introducing a new problem version, the *parity-counting problem version*, and a reduction from that to the decision version. The parity-counting problem is, as we remember from Chapter 3.1.1, to calculate the parity of the number of solutions (i.e. decide if the number is even or odd). This reduces to the decision version through the realisation that as long as at most one root of E has been *isolated*, meaning it is the only solution satisfying some condition, then an even amount of solutions directly imply that there is no root and an odd solution count implies the existence of a root. This isolation of solutions is achieved by Valiant-Vazirani affine hashing [VV86] (described in Chapter 2.5), which, in essence, consists of adding deliberately chosen linear equations to the system such that their common satisfiability implies that a solution to the extended system is unique (the condition on the solution that ensures it is isolated is thus that it satisfies all of the added linear equations). This following description of Björklund et al.'s algorithm is based on the one by Dinur [Din] and is kept brief both for convenience and because the similarities are large to Dinur's two algorithms, which we will study in more detail in the next two subchapters.

We start by noting that the parity of the number of roots is exactly $\sum_{\mathbf{x}^* \in \mathbb{F}_2^n} F(\mathbf{x}^*)$, when the sum is taken over $\mathbb{F}_2$ and 1 signifies odd and 0 signifies even. Similarily, the *partial parity* can be given as $G(\mathbf{y}) := \sum_{\mathbf{z}^* \in \mathbb{F}_2^k} F(\mathbf{y}, \mathbf{z}^*)$, and the corresponding *estimated partial parity* can be given as

$$\tilde{G}(\mathbf{y}) := \sum_{\mathbf{z}^* \in \mathbb{F}_2^k} \tilde{F}(\mathbf{y}, \mathbf{z}^*). \tag{41}$$

We wish now to relate the estimated partial parity to the true partial parity, for any $\mathbf{y}^* \in \mathbb{F}_2^{n-k}$, and we start by noting that the probability that the parities are equal is at least that of them being equal simply due to that their terms all are equal, that is

$$\mathbb{P}\left[\tilde{G}(\mathbf{y}^*) = G(\mathbf{y}^*)\right] = \mathbb{P}\left[\sum_{\mathbf{z}^* \in \mathbb{F}_2^k} \tilde{F}(\mathbf{y}^*, \mathbf{z}^*) = \sum_{\mathbf{z}^* \in \mathbb{F}_2^k} F(\mathbf{y}^*, \mathbf{z}^*)\right] \geq \mathbb{P}\left[\bigcap_{\mathbf{z}^* \in \mathbb{F}_2^k} \tilde{F}(\mathbf{y}^*, \mathbf{z}^*) = F(\mathbf{y}^*, \mathbf{z}^*)\right]. \tag{42}$$

Now by a standard union bound over $\mathbf{z}^*$, and by the relationship between F and $\tilde{F}$ given in Chapter 10.1, we arrive at

$$\mathbb{P}\left[\tilde{G}(\mathbf{y}^*) = G(\mathbf{y}^*)\right] \geq 1 - \sum_{\mathbf{z}^* \in \mathbb{F}_2^k} \mathbb{P}\left[\tilde{F}(\mathbf{y}^*, \mathbf{z}^*) \neq F(\mathbf{y}^*, \mathbf{z}^*)\right] \geq 1 - \frac{1}{2^{l-k}}. \tag{43}$$

[3]The relation follows from the observation that the monomials generated by going to $M(n, r+1, 2)$ from $M(n, r, 2)$ are those monomials of degree $r+1$ constructed by multiplying each monomial $\mathbf{x}^\alpha$ of degree r by some variable x_i such that $\alpha_i = 0$. Trivially, there is at most n admissable choices of i.

Thus one may recieve, from a choice of l, k, a known minimum probability that $\tilde{G}(\mathbf{y})$ correctly represents $G(\mathbf{y})$. Björklund et al. choose $l = k+2$, which results in that $\tilde{G}$ correctly calculates G for at least $\frac{3}{4}$ of the assignments. Then by iterating the process for new $\tilde{F}$ and regarding the majority vote between the instances as the predicted partial parity, this prediction is correct with probability exponentially close to 1 in n, if the number of repetitions of the procedure is lower bounded asymptotically by n. From the partial parities, the total parity is then directly computable by summing over the partial parities for all possible $\mathbf{y}^*$.

Except for the fact that one can utilise Valiant-Vazirani affine hashing to convert the decision problem to the parity-counting problem, Björklund et al. also make the critical note that evaluating $\tilde{G}$ may be done using self-recursion of the entire algorithm in itself since

$$\tilde{G}(\mathbf{y}^*) := \sum_{\mathbf{z}^* \in \mathbb{F}_2^k} \tilde{F}(\mathbf{y}^*, \mathbf{z}^*) := \sum_{\mathbf{z}^* \in \mathbb{F}_2^k} \prod_{1}^{l} (1 + R_i(\mathbf{y}^*, \mathbf{z}^*)).$$

This means that $\tilde{G}$ may for each $\mathbf{y}^*$ be evaluated by finding a solution to the parity-count problem of $\{R_1, ..., R_l\}$. It is also noted that one may interpolate $\tilde{G}(\mathbf{y})$ by the Möbius transform (outlined in Chapter 2.4) from only the evaluations of $\tilde{G}$ for $\mathbf{y} \in W_{dl-k}^{n-k}$, where we remember that $W_w^n := \{\mathbf{x} \in \mathbb{F}_2^n | HW(\mathbf{x}) \leq w\}$. This implies that interpolating $\tilde{G}(\mathbf{y})$ reduces to $\binom{n-k}{\downarrow(dl-k)}$ recursive calls to the parity-counting algorithm. This self-reduction reduces the exponential term in the time complexity since the sizes of the systems in the recursions are smaller than the original one. It is however understood that the self-recursion also births vast polynomial terms [Din21].

10.4 Dinur's first algorithm ($\texttt{Dinur}_1$), 2021

Correctness assumptions:	*None*
Worst-case time complexity:	$O^*(2^{(1-\frac{1}{2d})n})$ [Bit operations]
Space complexity:	$O^*(2^{(1-\frac{1}{2d})n})$ [Din21]

Idea: *The parity-counts of the subsystems are related and can thus be calculated more efficiently as a group than separately!*

In 2021, Itai Dinur continues the work of Björklund et al. by noting that the partial parities are significantly dependent on each other and may thus be computed more efficiently considered together rather than individually[Din]. This slight shift of approach is shown in that Dinur introduces yet another problem version, the *multiple parity-counting* problem, together with an algorithm for solving it. This new algorithm is called[4] $MultParityCount_{Dinur_1}$ and we will call the algorithm that solves PoSSo$_2$-Search using it $\texttt{Dinur}_1$, in order to differ between Dinur's first and second algorithm. The input to the new problem is the system E together with two parameters $k \leq n$ and $w \leq n-k$ and the output is the vector $V = \{V_0, ..., V_{\binom{n-k}{\downarrow w}-1}\} \in \mathbb{F}_2^{\binom{n-k}{\downarrow w}}$ such that

$$V_i = \sum_{\mathbf{z}^* \in \mathbb{F}_2^k} F(W_w^{n-k}[i], \mathbf{z}^*),$$

[4]The subscript was added by us to clarify that it belongs to Dinur's first algorithm rather than the second.

where $W_w^{n-k}[i]$ is the ith element of W_w^{n-k} under the natural ordering (that is, ordered as the exponents in lex, i.e. in increasing order if the bitstrings are interpreted as base 2 integers). Dinur introduces another splitting of the variables, by selecting a $k' \in \{1, ..., k\}$ and saying $\mathbf{u} := (z_1, ..., z_{k-k'})$, $\mathbf{v} := (z_{k-k'+1}, ..., z_k)$ and then $\mathbf{x} = (\mathbf{y}, \mathbf{u}, \mathbf{v})$. Similarily to above, $\tilde{G}$ is now defined as

$$\tilde{G}(\mathbf{y}, \mathbf{u}) := \sum_{\mathbf{v}^* \in \mathbb{F}_2^{k'}} \tilde{F}(\mathbf{y}, \mathbf{u}, \mathbf{v}^*), \tag{44}$$

and thus following from the analysis of Björklund et al.'s algorithm, where for each $(\mathbf{y}^*, \mathbf{u}^*)$ the estimated partial parity $\tilde{G}(\mathbf{y}^*, \mathbf{u}^*)$ is correct with probability $1 - \frac{1}{2^{l-k'}}$. Analogously, one chooses $l := k' + 2$ so that for each $(\mathbf{y}^*, \mathbf{u}^*)$, the probability that $\tilde{G}$ correctly estimates G is at least $\frac{3}{4}$.

Just as in Björklund et al.'s algorithm, the intention is now to interpolate $\tilde{G}(\mathbf{y}, \mathbf{u})$ from the assignments $(\mathbf{y}^*, \mathbf{u}^*) \in W_{dl-k'}^{n-k'}$. This is, however, the point at which the algorithms start to differ, in that Dinur notes that this interpolation may be handled by one recursive call to find the multiple parities on system $\{R_1, ..., R_l\}$, with parameters $k_{new} = k', w_{new} = dl - k'$. Now, once again, the partial parities can be obtained from evaluating $\tilde{G}$, repeating the whole procedure and predicting the partial parities according to a scoreboard.

In Algorithm 16 follows pseudocode for `Dinur`$_1$. It is a reformulation and extension of the one given in [Din]. Note that instead of feeding direct values of k and k' into the algorithm, we give parameters κ_0 and λ that describes what proportion of n that k should be and what proportion of k that k' should be, respectively. After the pseudocode of `Dinur`$_1$ follows the ones for subprocedures $Decision_{Dinur_1}$, which solves the decision problem version using the full parities, and $ParityCount_{Dinur_1}$, that solves the parity-counting problem by summing up the partial parities returned by $MultParityCount_{Dinur_1}$.

Algorithm 16 `Dinur`$_1$

Require: A set $E := \{P_1, ..., P_m\}$ of polynomials. $\lambda \in (0, 1)$ and $\kappa_0 \in (0, 1)$.

$Sol \leftarrow \emptyset$
if $Decision_{Dinur_1}(\{P_j(0, (x_2, ..., x_n))\}, \lambda, \kappa_0) = 1$ **then**
    $Sol \leftarrow (0)$
else if $Decision_{Dinur_1}(\{P_j(1, (x_2, ..., x_n))\}, \lambda, \kappa_0) = 1$ **then**
    $Sol \leftarrow (1)$
else
    Return $\emptyset$
end if
for $i \in \{2, ..., n\}$ **do**
    if $Decision_{Dinur_1}(\{P_j((Sol_1, ..., Sol_{i-1}, 0), (x_{i+1}, ..., x_n))\}, \lambda, \kappa_0) = 1$ **then**
        $Sol \leftarrow (Sol_1, ..., Sol_{i-1}, 0)$
    else if $Decision_{Dinur_1}(\{P_j((Sol_1, ..., Sol_{i-1}, 1), (x_{i+1}, ..., x_n))\}, \lambda, \kappa_0) = 1$ **then**
        $Sol \leftarrow (Sol_1, ..., Sol_{i-1}, 1)$
    else
        Return $\emptyset$
    end if
end for
Return A root Sol of E

Algorithm 17 $Decision_{Dinur_1}$

Require: A set $E := \{P_1, ..., P_m\}$ of polynomials. $\lambda \in (0,1)$ and $\kappa_0 \in (0,1)$.

for $c \in \{0, ..., n\}$ **do**
  $E' \leftarrow ValiantVazirani(E, c)$
  if $ParityCount_{Dinur_1}(E', \lambda, \kappa_0) = 1$ **then**
    Return 1
  end if
end for
Return 0

Algorithm 18 $ParityCount_{Dinur_1}$

Require: A set $E' := \{P_1, ..., P_{m'}\}$ of polynomials. $\lambda \in (0,1)$ and $\kappa_0 \in (0,1)$.

$k \leftarrow \lfloor \kappa_0 n \rfloor$
$\{V_1, ..., V_{2^{n-k}-1}\} \leftarrow MultParityCount_{Dinur_1}(E', k, n-k, \lambda)$
$Parity \leftarrow 0$
for $\mathbf{y}^* \in \mathbb{F}_2^{n-k}$ **do**
  $Parity \leftarrow Parity \oplus V_{\mathbf{y}^*}$
end for
Return $Parity$, the parity of the number of solutions of E'

Algorithm 19 $MultParityCount_{Dinur_1}$

Require: A set $E' := \{P_1, ..., P_{m'}\}$ of polynomials. Positive integers k, w and a $\lambda \in (0, 1)$.

1: $l \leftarrow k + 2$, $k' \leftarrow \lfloor k - \lambda n \rfloor$
2: **if** $k' \leq 0$ **then**
3: $\quad \{V_1, ..., V_{\binom{n-k'}{w}}\} \leftarrow BruteForceMultParity(E', k, w)$
4: **end if**
5: $Scoreboard \leftarrow (0, ..., 0)$ ▷ an array of $\binom{n-k}{w}2^{k-k'}$ zeros
6: $t \leftarrow 48n + 1$
7: **for** $iter \in \{1, ..., t\}$ **do**
8: $\quad \{R_1^{(iter)}, ..., R_l^{(iter)}\} \leftarrow GenerateR(E, l)$
9: $\quad V^{(iter)} \leftarrow MultParityCount(\{R_i((\mathbf{y}, \mathbf{u}), \mathbf{v})\}, k', dl - k', \lambda)$
10: $\quad \tilde{G}^{(iter)}(\mathbf{y}, \mathbf{u}) \leftarrow InterpolateGTilde(V^{(iter)})$
11: $\quad Truthtable\tilde{G} \leftarrow Evaluate(\tilde{G})$
12: $\quad$ **for** $\mathbf{y}^* \in W_w^{n-k} \times \mathbb{F}_2^{k-k'}$ **do**
13: $\quad\quad$ **if** $Truthtable\tilde{G}(\mathbf{y}^*) = 1$ **then**
14: $\quad\quad\quad Scoreboard(\mathbf{y}^*) \leftarrow Scoreboard(\mathbf{y}^*) + 1$
15: $\quad\quad$ **end if**
16: $\quad$ **end for**
17: **end for**
18: **for** $\mathbf{y}^* \in W_w^{n-k}$ **do**
19: $\quad Tally \leftarrow 0$
20: $\quad$ **for** $\mathbf{u}^* \in \mathbb{F}_2^{k-k'}$ **do**
21: $\quad\quad Tally \leftarrow Tally + Scoreboard((\mathbf{y}^*, \mathbf{u}^*))$
22: $\quad$ **end for**
23: $\quad$ **if** $Tally \geq 2^{k-k'-1}$ **then** ▷ A majority vote
24: $\quad\quad V_{\mathbf{y}^*} \leftarrow V_{\mathbf{y}^*} \oplus 1$
25: $\quad$ **end if**
26: **end for**
27: **Return** V

In Algorithm 19 above, $BruteForceMultParity$ (at line 3) is simply a brute-force version of $MultParityCount$ and $GenerateR$ (at line 8) is done according to the definition of R_i. $Interpolate$ and $Evaluate$ (lines 11 and 12 respectively) are both done via the Möbius transform and $ValiantVazirani$ (line 2 in Algorithm 17) is according to its definition (see Chapter 2.5). An important remark is that the Valiant-Vazirani hashing requires that one knows approximately the number of roots for the system, more precisely that there is between 2^c and 2^{c+1} roots, and since we generally do not, the entire parity count algorithm must be repeated for all such $c \in \{0, ..., n\}$. For more details on the subprocedures (and the algorithm in general) see [Din].

The main result of [Din] is stated below and the proof is sketched. The reader should note that Dinur does not state the computational model or a unit for his result. He does, however, directly refer to another complexity result of one of the used subfunctions (the Möbius transform) explicitly measured in the *number of bit operations* and thus we may well assume that this unit is intended to be used for the rest of Dinur's result as well.

Theorem 10.2 (Runtime of $\texttt{Dinur}_1$, reformulation of Theorems 1.1 and 3.1 in [Din])**.**
Given a polynomial system E, the probabilistic Algorithm 16 has a worst-case asymptotic time complexity of $O^*(2^{0.6943n})$ bit operations when $d = 2$ and $O^*(2^{(1-\frac{1}{2d})n})$ bit operations when $d > 2$.

Sketch of proof given in [Din].
We note that:

- The runtime of $\texttt{Dinur}_1$ is dominated by the calls to $Decision_{Dinur_1}$ and there are at most $2n$ of those.
- The number of equations added by $ValiantVazirani$ is logarithmic in c, an integer between 0 and n such that the number of roots to E is assumed between 2^c and 2^{c+1}. Thus the number of added equations are at most $O(\log_2(n))$, a change of the system that will be ignored by the O^*-notation.
- The runtime of $Decision_{Dinur_1}$ is dominated by the calls to $ParityCount_{Dinur_1}$, of which there are at most n.
- The runtime of $ParityCount_{Dinur_1}$ consists of the time of one execution of $MultParityCount_{Dinur_1}$ and 2^{n-k} additions. These two terms are inversely related to the chosen value of κ_0, in that the time needed for $MultParityCount_{Dinur_1}$ decreases as κ_0 decreases, increases as κ_0 increases, and the runtime of the additions behave the other way around. Therefore since both of the terms occupy the same range of possible sizes (essentially between $O(1)$ and $O(2^n)$), the two terms may be (and is) taken as equal by an optimal choice of κ_0.
- Denote by $T(k,w)$ the runtime of $MultParityCount_{Dinur_1}$, since n and d stay unchanged throughout the recursion, and let $T_{<subprocedure>}$ denote the runtime of said subprocedure. We have

$$T(k,w) = O\left(t \cdot \left(\underbrace{T(k', dl-k')}_{T_{MultParityCount(k',dl-k')}} + \underbrace{n \cdot \binom{n-k'}{\downarrow (dl-k')}}_{T_{Interpolate}} + \underbrace{n \cdot \binom{n-k}{\downarrow w} \cdot 2^{k-k'}}_{T_{Evaluate}}\right)\right),$$

since $T_{Evaluate}$ dominates all other steps in $MultParityCount_{Dinur_1}$. The second term stems from the interpolation, more precisely, from the interpolation of $\tilde{G}$ from its evaluations on $W_{dl-k'}^{n-k'}$ using the Möbius transform. Recall then from Chapter 2.4 that the complexity of such interpolation is $n\binom{n-k'}{\downarrow(dl-k')}$. The third term comes from evaluating $\tilde{G}$ on $W_w^{n-k'} \times \mathbb{F}_2^{k-k'}$ using the Möbius transform, and this term also dominates the complexity of all remaining steps in $MultParityCount_{Dinur_1}$.

From a somewhat technical reasoning, one then can guarantee a choice of λ, w and k such that the number of recursive calls becomes subexponential, and thus one can balance the terms in $T(k,w)$ to obtain an optimal parameter choice. This choice is the one indicated in the theorem, meaning that for $d > 2$ the total runtime of $\texttt{Dinur}_1$ is upper bounded asymptotically by

$$O^*\left(\underbrace{2n}_{\text{nr runs of } Decision_{Dinur_1}} \cdot \underbrace{n}_{\text{nr runs of } ParityCount_{Dinur_1}} \cdot \underbrace{2^{(1-\frac{1}{2d})n}}_{T(k,w)}\right) = O^*(2^{(1-\frac{1}{2d})n})$$

and analogously for $d = 2$. □

Correctness assumptions:	Assumption 1 (see below)
Assumptions for complexity:	Uniform randomness of E
Expected time complexity:	$O(n^2 \cdot 2^{(1-\frac{1}{2.7d})n})$ [Bit operations of SLP]
Space complexity:	$O(n^2 2^{(1-\frac{1}{1.35d})n})$

10.5 Dinur's second algorithm ($\texttt{Dinur}_2$), 2021

Dinur published another paper [Din21] in 2021 that adjusts $\texttt{Dinur}_1$ to obtain a more precise estimate of the time complexity, and to arrive at an algorithm with smaller polynomial factors. We call the new resulting algorithm $\texttt{Dinur}_2$. In [Din21], Dinur makes a few observations about the approach in his previous algorithm.

- **Observation 1:** The reductions from search to decision to parity-counting adds large polynomial factors to the runtime and they may be avoided entirely.
- **Observation 2:** One can isolate many solutions of E simultaneously by a variable partition.

Definition 10.1 (Isolated solutions [Din21])**.**
A solution $\mathbf{x}^* = (\mathbf{y}^*, \mathbf{z}^*)$ of E is called isolated with respect to the used variable partition if $(\mathbf{y}^*, \mathbf{z}')$ is not a root of E for all $\mathbf{z}'$ not equal to $\mathbf{z}^*$.

- **Observation 3:** All isolated solutions with respect to a partition $(\mathbf{y}, \mathbf{z})$ can be recovered bit-by-bit by computing the partial parities for each $\mathbf{y}^*$.

Proposition 10.1 (Reformulation of proposition 2.3 in [Din21])**.**
Let

$$G_0(\mathbf{y}) := \sum_{\mathbf{z}^* \in \mathbb{F}_2^k} F(\mathbf{y}, \mathbf{z}^*), \tag{45}$$

$$G_i(\mathbf{y}) := \sum_{\mathbf{z}^* \in \mathbb{F}_2^{k-1}} F(\mathbf{y}, z_1^*, ..., z_{i-1}^*, 0, z_{i+1}^*, ..., z_k^*), \tag{46}$$

with F defined as previously. Assume $(\mathbf{y}^*, \mathbf{z}^*)$ is an isolated solution to E with respect to the variable partition. Then

$$G_0(\mathbf{y}^*) = 1, \tag{47}$$

$$G_i(\mathbf{y}^*) = z_i^* \oplus 1 \forall i \in \{1, ..., k\}. \tag{48}$$

A direct consequence of this proposition is that $G_0(\mathbf{y}^*) = 1$ is a necessary condition for $(\mathbf{y}^*, \mathbf{z}^*)$ being an isolated solution and also one may obtain $\mathbf{z}^*$ entirely from the evaluations of $G_i(\mathbf{y}^*)$. Just as in Björklund et al.'s algorithm and $\texttt{Dinur}_1$, these G are inefficient to evaluate directly and therefore must be probabilistically approximated. Let $\tilde{E} := \{R_1, ..., R_l\}$ (as defined in Chapter 10.1) be the probabilistic representation of E. Also let $\tilde{G}_0$ and $\tilde{G}_i$ be probabilistic versions of G_0, G_i defined analogously but using $\tilde{F}$ instead of F.

- **Observation 4:** The set of all isolated solutions to $\tilde{E}$ for some partition forms a superset to the set of all solutions to E.

From the observations above, the overarching idea of $\texttt{Dinur}_2$ shines through:

Generate a system $\tilde{E}$, find isolated solutions to it by evaluating $\tilde{G}_0$, $\tilde{G}_i$ and test these candidate solutions on E.

Some remarks are to be made here. First of all, since the isolated solutions of $\tilde{E}$ are only of interest to us because they suggest isolated solutions of E, the procedure outlined above must be repeated a few times for new, independent, $\tilde{E}$. Secondly, an assumption for $\texttt{Dinur}_2$ to terminate correctly is:

- **Assumption 1:** *There is a high probability that there exists an isolated solution of E for the chosen partition.*

This assumption is needed because the algorithm, by design, cannot find any *unisolated* solutions. The justification of the assumption boils down to an argument about the randomness of the system, with the core being that since systems occuring in cryptography are ideally, from the designer's point of view, "random-looking" then the probability that a solution is not isolated with respect to a given partition is small. For a system generated uniformly at random, each assignment $\mathbf{x}^*$ is a root with a probability of around 2^{-m} and thus a solution to E should be isolated with probability around $1 - 2^{k-m}$. Thus Assumption 1 should reasonably hold if $m \gg k$ and the system is in essence as if generated uniformly at random and under the assumption that there is at least one root, which we obviously may assume for cryptographical reasons. Finally one should also note that testing each candidate solution of E is inefficient and therefore a candidate solution is tested on E first when it has been proposed for two different $\tilde{E}$.

Below in Algorithm 20 follows the pseudocode of $\texttt{Dinur}_2$, which is a reformulated version of the one in [Din21] to adopt our terminology.

Algorithm 20 $\texttt{Dinur}_2$

Require: A set $E := \{P_1, ..., P_m\}$ of polynomials. Two positive integers $k, d_{\tilde{F}}$.

$l \leftarrow k+1$, $w \leftarrow d_{\tilde{F}} - k$
$iter \leftarrow 0$
while $True$ **do**
  $iter \leftarrow iter + 1$
  $\tilde{E}^{(iter)} \leftarrow GenerateETilde(E, l)$
  $CurrPotSols \leftarrow OutputPotSols(\tilde{E}^{(iter)}, k, w)$
  $PotSolsList_{iter} \leftarrow CurrPotSols$
  for $\mathbf{y}^* \in \mathbb{F}_2^{n-k}$ **do**
    if $CurrPotSols_{\mathbf{y}^*0} = 1$ **then**
      for $iter_{old} \in \{1, ..., iter-1\}$ **do**
        if $CurrPotSols_{\mathbf{y}^*} = PotSolsList_{iter_{old}, \mathbf{y}^*}$ **then**
          $CandidateSol \leftarrow (\mathbf{y}^*, CurrPotSols_{\mathbf{y}^*1}, ..., CurrPotSols_{\mathbf{y}^*k})$
          if $E(CandidateSol) = 0$ **then**
            Return $CandidateSol$
          end if
        end if
      end for
    end if
  end for
end while

Algorithm 21 $OutputPotSols$

Require: A set $\tilde{E} := \{R_1, ..., R_l\}$ of polynomials. Two positive integers k, w.

$(\tilde{GValues}_0, ..., \tilde{GValues}_k) \leftarrow ComputeGTildeValues(\tilde{E}, k, w)$
$\tilde{G}_0 \leftarrow Interpolate(\tilde{GValues}_0)$ ▷ Interpolation on W_w^{n-k}.
for $i \in \{1, ..., k\}$ **do**
  $\tilde{G}_i \leftarrow Interpolate(\tilde{GValues}_i)$ ▷ Interpolation on W_{w+1}^{n-k}.
end for
$Evals \leftarrow ((0, ..., 0), ..., (0, ..., 0))$ ▷ A $k+1$-length array of 2^{n-k} - length arrays of zeros.
for $i \in \{1, ..., k\}$ **do**
  $Evals_i \leftarrow Evaluate(\tilde{G}_i)$
end for
$Out \leftarrow ((0, ..., 0), ..., (0, ..., 0))$ ▷ A 2^{n-k}-length array of $k+1$ - length arrays of zeros.
for $\mathbf{y}^* \in \mathbb{F}_2^{n-k}$ **do**
  if $Evals_{0\mathbf{y}^*} = 1$ **then**
    $Out_{\mathbf{y}^*0} \leftarrow 1$
    for $i \in \{1, ..., k\}$ **do**
      $Out_{\mathbf{y}^*i} \leftarrow Evals_{i\mathbf{y}^*} + 1$
    end for
  end if
end for
Return Out

Algorithm 22 *ComputeGTildeValues*

Require: A set $\tilde{E} := \{R_1, ..., R_l\}$ of polynomials. Two positive integers k, w.
1: $(Sols_1, ..., Sols_L) \leftarrow BruteForceSystem(\tilde{E}, n-k, w+1)$
2: $(\tilde{G}Values_0, ..., \tilde{G}Values_k) \leftarrow ((0, ..., 0), ..., (0, ..., 0))$ ▷ A k-length array of $|W_w^{n-k}|$ - length arrays of zeros.
3: **for** $(\mathbf{y}^*, \mathbf{z}^*) \in Sols$ **do**
4: **if** $HW(\mathbf{y}^*) \leq w$ **then**
5: $index \leftarrow IndexOfInW(\mathbf{y}^*, n-k, w)$
6: $\tilde{G}Values_{0,index} \leftarrow \tilde{G}Values_{0,index} + 1$
7: **end if**
8: **for** $i \in \{1, ..., k\}$ **do**
9: **if** $z_i^* = 0$ **then**
10: $index \leftarrow IndexOfInW(\mathbf{y}^*, n-k, w)$
11: $\tilde{G}Values_{i,index} \leftarrow \tilde{G}Values_{i,index} + 1$
12: **end if**
13: **end for**
14: **end for**
15: **Return** $(\tilde{G}Values_0, ..., \tilde{G}Values_k)$

The subprocedure *GenerateETilde* is done directly according to the definition of $\tilde{E}$. $BruteForceSystem(\tilde{E}, k', w')$ performs `FES`-like exhaustive search on all values in $W_{w'}^{k'} \times \mathbb{F}_2^{k'}$. *Interpolate* and *Evaluate* are both performed using the Möbius transform, similarily to in Björklund et al's algorithm and $\texttt{Dinur}_1$. Below follows a theorem of the *expected* runtime of $\texttt{Dinur}_2$ and a sketch of a proof. For more details of the algorithm at large, and a more thorough runtime analysis, see [Din21].

Theorem 10.3 (Reformulation of Theorem 4.1 and Chapter 4.2 in [Din21])**.**
Given a system E, $\texttt{Dinur}_2$ is, under Assumption 1, expected to terminate in $O(n^2 2^{0.815n})$ time for $d = 2$ and $O(n^2 2^{(1-\frac{1}{2.7d})n})$ time for $d > 2$ (as a bitwise-SLP).

Proof-sketch: Sketch of runtime analysis in [Din21].
Assume that the expected runtime for testing a proposed solution of E is at most $kn2^k$ bit operations, that $m \geq 2k + 4$ and that Assumption 1 above holds. Then the expected runtime of $\texttt{Dinur}_2$, denoted by T, satisfies

$$T \leq 4\left(2d \cdot \log_2(n) 2^k \cdot \binom{n-k}{\downarrow (kd - k + d + 1)}\right) + kn2^{n-k}. \tag{49}$$

The first term stems from the "largest" execution of $BruteForceSystem$, which performs `FES`-like exhaustive search on $W_{w+1}^{n-k} \times \mathbb{F}_2^k$ and thus takes $O(d \log_2(n)(2^k \cdot |W_{w+1}^{n-k}|))$ bit operations (see the runtime of `FES` in Chapter 7). The second is an upper bound of the runtime for $Evaluate(\tilde{G}_i)$, i.e. evaluating all $k+1$ of $\tilde{G}_i$ on $\mathbb{F}_2^{n-k}$ via the Möbius transform, which for each function takes at most $(n-k)2^{n-k}$ bit operations (see Chapter 2.4). The runtimes of all other steps are dominated by these two, most interestingly is the interpolation of $\tilde{G}$ dominated by its evaluations since the interpolation of $\tilde{G}_0$ (in its "largest iteration") is done on W_w^{n-k} rather than on W_{w+1}^{n-k}, as the evaluation is. The estimate given in (49) can be rewritten as

$$T \leq O\left(\log_2(n) 2^k \binom{n-k}{\downarrow (kd - k + d + 1)}\right) + O(n^2 2^{n-k}). \tag{50}$$

Just as in most previous runtime proofs in this thesis is the final step now to reformulate the two terms into similar forms and then balance them asymptotically by parameter choices. Since the first term

is of a more complicated formula, we will upper bound it by something that is similar to the second term.

First of all do we, if $k(d-1)+d+1 \leq (k+2)(d-1)+1$ and $2(k+2)(d-1) \leq n-k$ (they hold if $d \geq 2$ and roughly $n \gg kd$), have that the binomial coefficient in the first term is upper bounded by

$$\binom{n-k}{\downarrow (kd-k+d+1)} = O\left(n\sqrt{n}\binom{n-k-2}{(k+2)(d-1)}\right). \tag{51}$$

Further, through that $\binom{a}{b} \leq 2^{aH(\frac{b}{a})}$ for any two integers a, b, we arrive at that the first term may be upper bounded by

$$O(n\log_2(n)\sqrt{n}2^{k+2}2^{(n-k-2)H(\frac{(k+2)(d-1)}{n-k-2})}) = O(n^2 2^{nH(\frac{(k+2)(d-1)}{n-k-2})}), \tag{52}$$

where $H(a) := -a\log_2(a) - (1-a)\log_2(1-a)$ for $a \in (0,1)$ is the *binary entropy* function. Then, if we define a $0 < \gamma < 1$ such that $\gamma := \frac{k+2}{n}$, and choose a γ (and through that choose k) so that $H(\frac{\gamma(d-1)}{1-\gamma}) \leq 1 - \frac{\gamma}{1-\gamma}$, then the two terms are balanced and the total runtime of the algorithm become

$$O(n^2 2^{(1-\gamma)n}). \tag{53}$$

More specifically, a choice of $\gamma = \frac{1}{2.7d}$ satisfies all demands on the parameter. Note also that it is demanded that k be an integer, and thus for a fixed value of n, there is limited precision in the choice of γ, since valid choices of γ are dictated by k and n. However since this analysis is done asymptotically in n and the error in γ from demanding k to be an integer tends to 0 as n grows, the given formula on the complexity holds. From this follow the theorem.

□

Roughly speaking, one can summarise the demands stated in [Din21] on the parameters more or less by that d must be small compared to n and $2^k \gg n$.

In Figure 5 below is an illustration of the behaviour of $\texttt{Dinur}_2$ for one fictive example run of the algorithm. The basis of the figure is the input domain, $\mathbb{F}_2^n$, and the set of roots of E, denoted by S. On top there is, for three iterations, the true solutions to the randomly generated polynomial systems $\tilde{E}_i$ (if i is the iteration number), denoted by $\tilde{S}_i$ and also the sets of proposed potential solutions, denoted by $pSols_i$. One key difference between the behaviour of this algorithm as compared with, for example, `Crossbred` (see its illustration in Chapter 9.3) is that $\texttt{Dinur}_2$ does not test candidate solutions before they have been proposed twice, meaning that for the example in the illustration, there is no testing of candidate solutions before the third iteration.

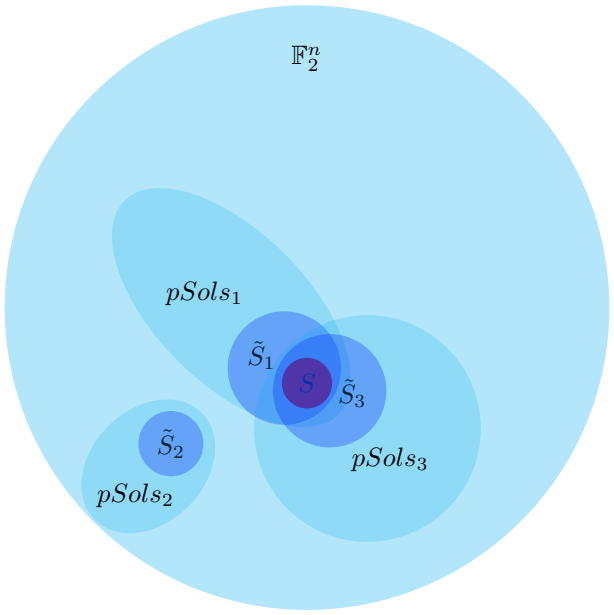

Figure 5: Illustration of the input space, solution sets and candidate solution sets for an example run of $\texttt{Dinur}_2$.

11 Intermezzo - Implementing $\texttt{Dinur}_2$

Since $\texttt{Dinur}_2$ was introduced only recently and has a very impressing theoretical performance, implementing it to test it experimentally is a highly exciting scholarly pursuit. The only supposed implementation of the algorithm at the time of writing is the one presented in the 2021 research article by Barbero et al. [Bar+21]. A closer look, however, at the implementation given in [Bar+21] reveals some key differences to the specifications of the algorithm as it is given by Dinur in [Din21]:

1. The implementation by Barbero et al. does only admit quadratic polynomial systems. Whilst this is not explicitly stated in their paper, it may be implicitly understood since it is clearly stated that all of their analyses concern such systems. The choice to restrict the implementation to quadratics does of course not invalidate their findings per se, but it should be noted that since Dinur clearly points out the merit of his algorithm for higher degree systems as well (he does, for example, highlight the case when $d = 4$) then this restriction is to be seen as significant.

2. The generation of random systems in the implementation is not according to the instructions by Dinur (chapter A.2 in [Din21]). Specifically, there is no fixing of a solution in the randomly generated systems and thus there is no guarantee that the generated problem instances actually have at least one solution. This does not affect the implementation of the algorithm in itself but since the distribution of input systems differ from the one in [Din21] then one must bear in mind that the expected results are not necessarily the same.

3. The bruteforce part of the algorithm (the procedure we call $BruteForceSystem$) is supposed to be `FES`-like, meaning in essence that the enumeration of the input space should be done efficiently (for example by using Gray codes) and that the early-abort strategy should be used. Barbero et al. rather implement a naive version of bruteforce. This difference, as opposed to the two previously perhaps, is actually critical since the estimate of the analysis of $\texttt{Dinur}_2$ relies on that the complexity of the bruteforce subprocedure can be assumed to be in essence that of `FES`, which is not the case for the version implemented.

4. The implementation of the Möbius transform used by Barbero et al. is a standard in place version. Dinur however prescribes that one should use a memory-optimised recursive version instead to decrease the space complexity of the algorithm. The choice to keep the iterative implementation should not affect the time complexity but the space complexity of the implemented algorithm should be expected to be significantly higher than the one of $\texttt{Dinur}_2$.

The differences between Dinur's description of his algorithm and the implementation by Barbero et al., especially regarding the bruteforce procedure, are substantial enough that one should question how well the experimental results reported in [Bar+21] reflect the performance of $\texttt{Dinur}_2$. To the defence of Barbero et al. one must say that they make it very clear that their implementation has not been optimised (although not excusing the absence of a discussion on the significant changes to subprocedures) and that their implementation is close enough to Dinur's description to provide a stepping stone for further development (which was stated as a key contribution of their implementation).

To provide a more representative implementation of $\texttt{Dinur}_2$ and further explore the properties of the algorithm, we extend the existing implementation. Our implementation may be found at https://github.com/FredrikMeisingseth/FredrikMeisingseth-Masters_thesis_implementing_Dinur2

11.1 An extended implementation

As the title of this subchapter implies, we would like to make it clear that the implementation we provide must be seen as an extension of the one given by Barbero et al. and that the changes made by us constitute only a small part of the implementation in total. We also remark that we do not consider our version of the implementation as complete either, primarily because the bruteforce step is still not sufficiently `FES`-like for its performance to be conjectured to be essentially that of `FES`.

11.1.1 Admitting arbitrary degree systems

Updating the implementation to allow polynomials of arbitrary degree consisted mainly of changing the supportive code defining the polynomial type so that its representation of a polynomial (and related functions) no longer are hard-coded for the case of quadratics. This process was quite straight-forward but does contain the quite interesting choice of how one should represent a polynomial.

The implementation by Barbero et al. represents a quadratic polynomial compactly by the coefficients $a_{ij}, b \in \mathbb{F}_2$ according to the formula

$$P(\mathbf{x}) := \sum_{1 \leq i \leq j \leq n} a_{ij} x_i x_j + b, \tag{54}$$

which means that the polynomial is well defined by the $\frac{n(n+1)}{2} + 1$ bits in a_{ij} and b since, as we remember, $x_i^2 = x_i$. This representation is memory-efficient, since it requires much less space than the standard way of defining a polynomial by the value of each of its 2^n coefficients through the formula

$$P(\mathbf{x}) := \sum_{\mathbf{u} \in \mathbb{F}_2^n} \alpha_{\mathbf{u}} \mathbf{x}^{\mathbf{u}}, \tag{55}$$

as described in Chapter 2.4 about the Möbius transform. For simplicity (mainly due to that there already existed a convenient representation of a boolean function), we used this standard representation but we will remark further that it should ideally be changed.

The key observation leading to the desire to not use the standard representation is that when $d \ll n$ then almost all of the coefficents are zero since they correspond to a monomial with degree over d. For example, with $n = 10, d = 2$ then at most $\frac{\binom{10}{2}+\binom{10}{1}+\binom{10}{0}}{2^{10}} \approx 5\%$ of the coefficents are non-zero and for $n = 20, d = 2$ then only $\frac{\binom{20}{2}+\binom{20}{1}+\binom{20}{0}}{2^{20}} \approx 0.02\%$ of the coefficents are possibly non-zero. Thus one must conclude that there is much storage space to be saved if one manages to represent a polynomial in a way such that the coefficents representing polynomials of a degree larger than d are not stored.

11.1.2 Generating systems with pre-fixed solution

The system generation implemented by Barbero et al. consists of each polynomial being uniformly randomly generated, in the sense that each coefficient (corresponding to a monomial of admissable degree) is assigned to either 0 or 1 with equal probability. Whilst this is a natural way of defining a random system, it is not quite the same as the definition used by Dinur. He rather specifies that his analysis is done under the assumption "that the polynomials of E are chosen independently and uniformly at random from all degree d polynomials, conditioned on having a pre-fixed solution ... chosen initially independently of E" [Din21]. The difference in these two choices regarding system generation

is small but since the change needed to conform with Dinur's instructions on how to generate random systems is only a few lines of code, then we make this change according to Chapter A.2 in [Din21].

The main argument for making this change is to get closer to the settings of Dinur's analysis of his algorithm but another advantage of generating polynomials this way is that it guarantees the existence of a root even for overdetermined systems, where the previous method of system generation would, with high probability, result in systems without a solution.

11.1.3 Making the bruteforce FES-like

As noted on in the introduction of this chapter, the `FES`-like bruteforce part of $\texttt{Dinur}_2$ is replaced by Barbero et al. by a naive bruteforce and thus the resulting algorithm is significantly slower than $\texttt{Dinur}_2$ theoretically. More specifically, the analysis of the runtime that is summarised in the proof-sketch of Theorem 10.3 above must be changed according to the fact that naive exhaustive search, on m polynomials, requires $O(m\binom{n}{d})$ bit operations per datapoint in the search space rather than the expected $O(d \cdot \log_2(n))$ needed by `FES`.

The changes needed to turn the bruteforce subprocedure by Barbero et al. into something with the same expected runtime as `FES` is, firstly, to implement the early-abort strategy and, secondly, to implement the algorithm for finding all roots of one polynomial using only $O(d2^n)$ bit operations as outlined in Chapter 4 of [Bou+10]. We did implement the early-abort strategy but due to time constraints we did not implement the fast algorithm for finding all roots of one single polynomial. The code for our adapted version of `FES` is, however, structured such that replacing the current naive way of enumerating the input space with the intended version should be reasonably effortless. Thus we remark that the bruteforce in our implementation is still not `FES`-like in the sense of performance but it is slightly closer to `FES` with regards to code structure.

11.1.4 Updating the Möbius transform implementation

Due to time constraints, we did not change the implementation of the Möbius transform to a recursive version as prescribed by Dinur.

11.2 Experimental results

Even though we do not consider our implementation a complete implementation of $\texttt{Dinur}_2$, we can still examine a few properties of the implementation. Below follows first an investigation into the runtime of our implementation. It has the two goals of estimating a formula for the runtime and of investigating if the runtime is essentially agnostic to the relationship between m and n, as is the case asymptotically for $\texttt{Dinur}_2$, as well as its dependence on the choice of d. Secondly, we investigate the runtime of Barbero et al.'s implementation in order to seek confirmation of the results in [Bar+21].

All of the code executions are done on a mid-range gaming laptop, see specifications in Table 6 below. All runtimes are recorded by hyperfine on a noisy system, which means that all measurments must be seen as quite rough. The parameter choices are, for both of the implementations, fixed at the values proposed by Dinur's complexity analysis, which means that they are likely not optimal for the ranges of system parameters that are used, since the parameter choices are derived to optimise the asymptotic complexity.

CPU	Intel Core i7-976H, 2.60Hz
RAM	8GB DDR4
OS	Linux (Ubuntu 20.04) subsystem for Windows

Table 6: Table over hardware specifications of the machine used for code execution.

11.2.1 Investigating the runtime of our implementation

In order to estimate a formula for our implementation, we randomly generate systems for n between 10 and 18 for $m = n, m = 2n, m = \frac{n}{2}$ and for $d = 2, d = 4, d = 8$, run the algorithm on the systems, record the runtimes (as the average runtime of 10 executions) and then perform linear regression on the runtimes normalised under the assumption of some formula for the complexity. The choice of these specific parameters were ruled mainly by the considerable runtime of our implementation. The hypothesis, stemming from the hope that the runtime should be similar to that of $\texttt{Dinur}_2$, is that the runtime will be essentially unaffected by the relationship between m and n, increase exponentially as a function of n and that the runtime should increase as d grows but only barely.

In Figure 6 we see the measured runtimes of our implementation. From the figure it is clear that the runtime increases exponentially as a function of n. Also, there does not appear to be any noticable impact on the runtime from the choice of d or the relationship between m and n, although this may simply be due to the inaccuracy of the measurements, an insufficient amount of data or due to that the effect of these factors are negligible in relationship to that of the change in n. Another source of inaccuracy might be that each of the independent executions are performed in the same polynomial system, and thus could a sudden drop (or increase) in runtime possibly be due to that specific input for those parameters being more (or less) favourable to solve than those of the previous n. The choice to only generate one system for each parameter set was made to decrease the amount of data to be stored (as all data used is available in the git repo).

The methodology chosen to estimate a formula for the runtime is to make one of two assumptions, either that the runtime is (i) of the form $a \cdot 2^{bn}$ or (ii) that the runtime is of the form $a \cdot n^2 2^{bn}$ for some coefficients $a, b \in \mathbb{R}$. These specific choices formulas are used since the first one purely an exponential runtime, and may thus serve as a reasonable null hypothesis of sorts, and the second formula is chosen since it is of the form that the complexity of $\texttt{Dinur}_2$ has. Under each of these assumptions of the runtime formula we re-scale the recorded runtimes, meaning that for (i)

$$T(n) \approx a \cdot 2^{bn} \implies \log_2(T(n)) \approx \log_2(a2^{bn}) = \log_2(a) + bn,$$

where $T(n)$ is the function we will estimate the runtime as, and thus the logarithmised data points are estimations of $log_2(T(n))$. Similarily for (ii)

$$T(n) \approx a \cdot n^2 \cdot 2^{bn} \implies \log_2(\frac{T(n)}{n^2}) \approx \log_2(a) + bn.$$

The respective coefficients a, b are estimated using standard linear regression. We use the python package *scikit-learn*[Ped+11] for this purpose, with the loss function being the residual sum of squares and with uniform weights for the samples.

This methodology does not only result in a estimate of the (expected) complexity function but also allows us to evaluate (roughly) how well the recorded runtimes correspond to the assumption of the formula of the runtime function, and this is done by some metric of how well the sample datapoints correspond to the corresponding predictions by the regression model. Although there are many possible such measures, we choose to use the R^2-*score* (the coefficient of determination) due to it being readily

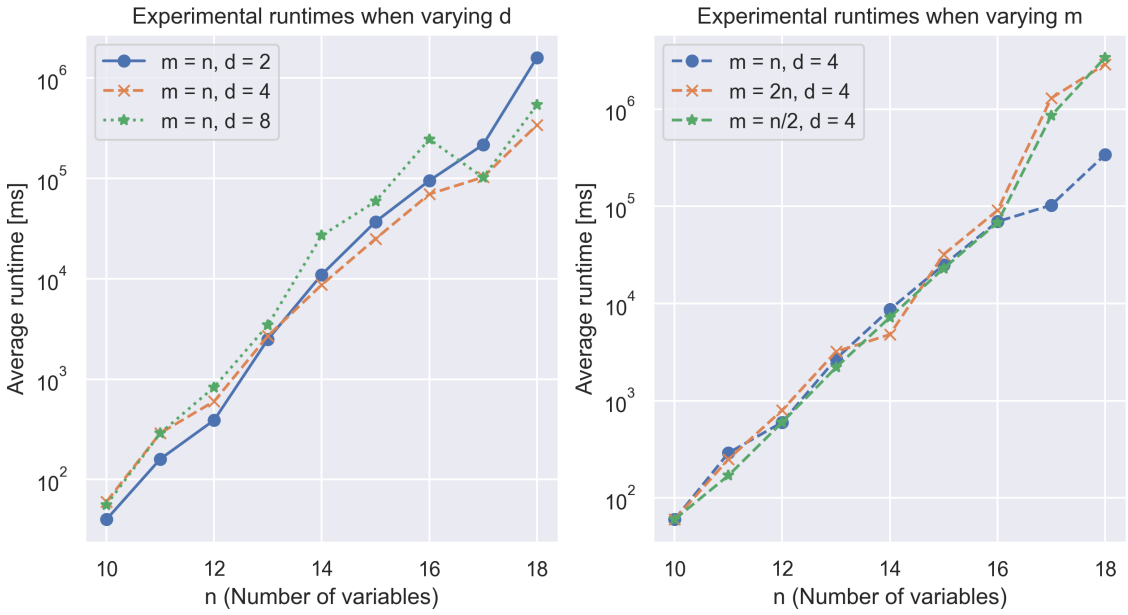

Figure 6: Recorded runtimes for our implementation as a function of n, with varying d or m respectively. The reported runtime is the average runtime of 10 repetitions of the program, where each execution solves the same pre-generated system.

available for the linear regression class in scikit-learn. Quite simplified, the R^2-score measures how well the model fits the observed runtimes, with a perfect score being 1 and a arbitrarily low worst possible score. This means that by comparing the two R^2-scores of the models resulting from assumption (i) and (ii), we may argue about whether or not one of the assumed formulas is a better estimate of the runtime of the algorithm. One must however note that this approach is quite rough and that even if one of the formulas has a very high R^2-score then this is very much not enough to conclude that said formula is a good representation of the runtime of the implementation generally, but rather it is to be seen as an indicator of wether or not the formula is a reasonable rough estimate of the complexity for the specific magnitude of parameters.

Although one could have opted to make the regression more complicated, in order to let it consider also m and d, we preferred the simplicity of the formulas (i) and (ii) and thus the regressions we make are restricted to explicit choices of these parameters. Since we do not expect the specific choice to affect the results of the analysis significantly (based on the result from the analysis of parameter choice), we choose, somewhat arbitrarily, to perform the runtime analysis for the runtimes when $m = n, d = 4$. The results were that for (i): $\log_2(a) \approx -8.9 \implies a \approx 2^{-8.9}$ and $b \approx 1.54$, which means

$$T(n) \approx 2^{1.54n-8.9}.$$

The results for (ii) was: $\log_2(a) \approx -13.5$ and $b \approx 1.33$ and thus

$$T(n) \approx n^2 2^{1.33n-13.5}.$$

The re-scaled samples and their respective regression line may be seen in Figure 7.

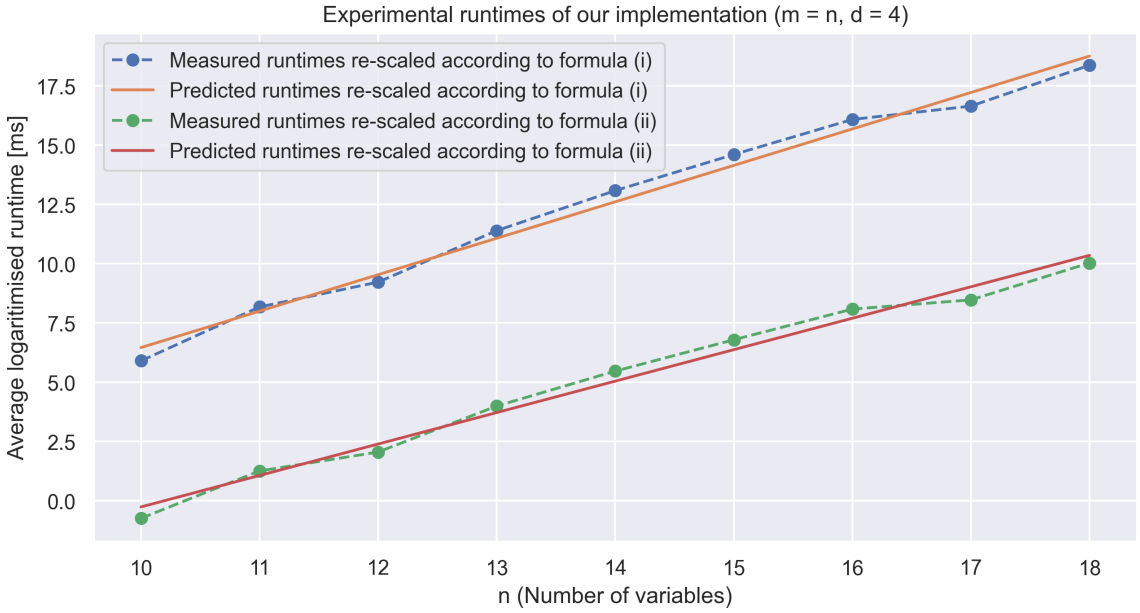

Figure 7: Runtimes after having been re-scaled under one of two assumptions of a formula for the complexity as well as the regression lines fitted to two respective sets of samples.

The R^2-scores for the regressions were 0.985 and 0.979 respectively. From this we may draw the conclusion that there is no meaningful indication that one should consider (ii) a more faithful formula for the runtime of our implementation than (i) and also, since the R^2-scores are so high, one should reasonably discard any hypothesis that the asymptotic complexity of the algorithm we implement should have an exponent smaller than $1.0 \cdot n$, i.e. a complexity with an exponential asymptotic speed-up over exhaustive search.

11.2.2 Investigating the runtime of Barbero et al.'s implementation

Similarily to the previous subchapter, we here record runtimes of Barbero et al.'s implementation, re-scale the samples, perform linear regression on them and analyse the resulting estimated runtime function. This investigation is however done under quite a few caveats, mostly regarding previously mentioned flaws in their implementation but also, perhaps most critically, regarding the way that they have chosen to test their implementation. The two main problems here are, as mentioned before, that there is no fixed solution for the generated system and that the algorithm does not loop indefinitely (or very long) if no solution is found. These problems mean that, as opposed to in our implementation (and in `Dinur`$_2$), one could affect the runtime of the algorithm by setting the maximum iteration count somewhat low and thus decrease the estimated complexity by exploiting that systems without solution are included in the test but quite quickly abandoned rather than the algorithm continuing its search for a root that is not there.

Regardless of the fact that a comparison between their implementation and ours is not directly meaningful, we may by investigating the runtime of their implementation attempt to confirm some of the experimental claims of Barbero et al., most importantly their result that the growth rate (by which we assume they mean the average ratio between measured runtimes for adjacent values of n, i.e. $\frac{T(n+1)}{T(n)}$) of their implementation is $2^{0.818}$. We also investigate whether we may find an indication that one

should conjecture that the complexity of their implementation is according to (ii) rather than (i). Just as Barbero et al., we run their algorithm for square quadratic systems for n between 14 and 30. The recorded runtimes may be seen in Figure 8.

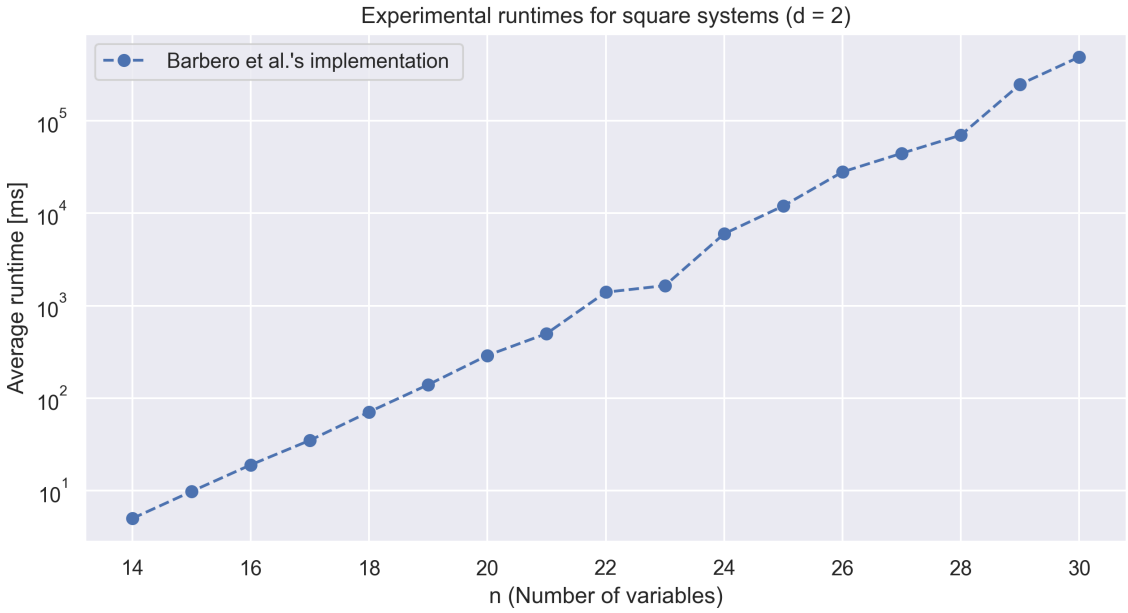

Figure 8: Recorded runtimes for Barbero et al.'s implementation as a function of n. The reported runtime is the average (amortised) runtime of 10 repetitions of the program, where each execution generates and solves 10 systems.

The experimental growth rate is calculated and shown to be $2.14 \approx 2^{1.09}$, thus contradicting the growth rate claimed in [Bar+21].

Then the recorded runtimes are re-scaled according to formulas (i) and (ii) as in the previous subchapter and the results are plotted, as well as the line fitted by the linear regression, in Figure 9. The regression outputs $\log_2(a) = -12.4$ and $b = 1.03$ for formula (i) and $\log_2(a) = -18.3$ and $b = 0.899$ for formula (ii). The R^2-scores of the regressions were 0.997 and 0.995 respectively and thus, similarily to the analysis of our implementation, one must draw the conclusion that the measured runtimes do not meaningfully indicate that the implementation exhibits a complexity particularily similar to that of `Dinur`$_2$ (as opposed to a complexity of formula (i), for example).

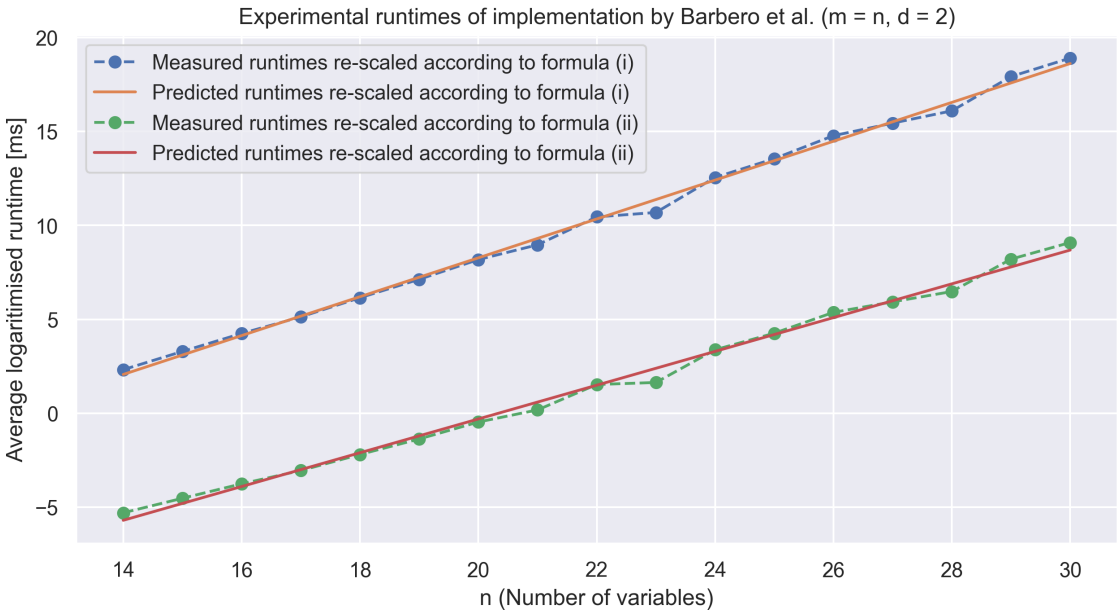

Figure 9: Runtimes after having been re-scaled under one of two assumptions of a formula for the complexity as well as the regression lines fitted to two respective sets of samples.

We should note that whilst we are not able to reproduce the results by Barbero et al., this could theoretically be due to us failing to faithfully reconstruct their experiments. For example, Barbero et al. claim to run their experiments on systems confirmed to have one unique solution whereas we use their test-script, which gives no such guarantees. However, as we have analytically argued before, one should not *expect* their implementation to have an experimental runtime very close to the asymptotical complexity of $\texttt{Dinur}_2$ (partly due to the considerable differences in the algorithms but also due to the fact that the complexity given by Dinur is *asymptotical*) and thus one must consider the strength of Barbero et al.'s complexity claims questionable at best.

12 Performance comparisons

In this chapter we will compare the algorithms surveyed in Chapters 7-10 according to the directions of Chapter 4. The comparison starts by considering what problem versions the different algorithms solve and what models of computation are used to define time complexity. An overview can be seen in Table 7 below.

Algorithm [paper]	**Problem version**	**Time complexity unit**
`FES` [Bou+10]	Exhaustive	"Bit operations"
F_4 [Fau99]	Exhaustive	*unspecified*
`Hybrid-`F_5 [BFP09]	Search	"Basic operations"
`XL` [Cou+00]	Search	*unspecified*
`BDT` [BDT21]	Exhaustive ($\mathcal{MQ}$)	"Operations"
BooleanSolve [Bar+13]	Exhaustive ($\mathcal{MQ}$)	"Arithmetic operations"
`Crossbred` [JV18]	Exhaustive	*unspecified*
Lokshtanov et al. [Lok+]	Decision	`RAM` model
Björklund et al.[BKW19]	Parity-count	*unspecified*
$\texttt{Dinur}_1$[Din]	Multiple parity-count	"Bit operations"
$\texttt{Dinur}_2$[Din21]	Search	Bit operations of SLP

Table 7: A table over problem versions solved by the algorithms and the model of computation used to derive complexity results.

From Table 7 above one can see that there is a large divide between algorithms solving the exhaustive version of PoSSo and those solving the search version. Since we are primarily interested in the search version, this should be interpreted as a slight disadvantage for the algorithms solving the exhaustive version since they are, as compared with PoSSo-Search, solving a more demanding problem. It is also clear from Table 7 that there is little consistency in the choices of computational model and unit. This will be resolved by first restricting our comparisons to just a few algorithms through arguments of reduncancy and then comparing situations in which the remaining algorithms perform more (or less) well.

Below in Table 8 we see an overview on the assumptions made by each algorithm. In it we can see that some assumptions are quite common, such as that of semi-regularity and those whose essence is that the polynomial system "looks random".

Algorithm [paper]	**Assumptions for complexity result**
`FES` [Bou+10]	Uniform randomness of system
F_4 [Fau99]	*None*
`Hybrid-`F_5 [BFP09]	Semi-regularity.
`XL` [Cou+00]	$d = 2$, $D \ll n$
`BDT` [BDT21]	Heuristic randomness assumptions
BooleanSolve [Bar+13]	Heuristic assumptions regarding semi-regularity
`Crossbred` [JV18]	Semi-regularity
Lokshtanov et al. [Lok+]	*None*
Björklund et al.[BKW19]	*None*
$\texttt{Dinur}_1$[Din]	*None*
$\texttt{Dinur}_2$[Din21]	Uniform randomness of system

Table 8: A table over the assumptions made by respective algorithm.

12.1 Redundancy arguments

In order to make the upcoming in-depth comparisons more concise, we will now argue why some of the algorithms presented are to be seen, in general, as a less effective version of some other and are thus to be regarded as not constituting a significant part of the state-of-the-art.

Faugére's methods: We regard F_5 as a specialised version of `Hybrid-`F_5 for obvious reasons. We will also, for less obvious reasons, regard F_4 as eclipsed by F_5, since the complexity results for them both are, in essence, the same (i.e. based on the analysis that the runtime is dominated by reducing the largest matrix) and their author claims the superiority of F_5 [Fau02].

The `XL` family: As above, since `XL` is a subcase of `FXL`, the original `XL` algorithm may be disregarded. Also, we do follow the redundancy argumentation of [Ars+04] and [Alb+12] and regard the entire `XL` family, in essence, as versions of Faugére's algorithms and therefore argue that these share the same position in the state of the art. One must however note that there are results that point to that the relationship between the two families are not as straight-forward as [Ars+04] and [Alb+12] might have you think, see for example the claim that `WXL` has a far lower memory complexity than F_4 [Moh+10].

`BDT` and BooleanSolve: Both of these are deemed redundant primarily since, as far as we know, there are no papers directly extending the algorithms to enable them to solve over-quadratic systems (since both `FXL` and `Crossbred` are considered separately). Also, one could argue that `BDT` is to be seen as a subcase of `Crossbred` and is thus to be disregarded directly. Finally, the role that BooleanSolve have held in the state-of-the-art for solving $\mathcal{MQ}$, namely that it has the lowest claimed asymptotic complexity, has been taken over by $\texttt{Dinur}_1$, which has a lower complexity claim and relies on no assumptions.

The polynomial method: The algorithms of Lokshtanov et al. and Björklund et al. are to be seen as entirely eclipsed by the algorithms of Dinur. From the perspective of striving for an optimally low exponent, then $\texttt{Dinur}_1$ is the best, and from the perspective of having a complexity estimate fine-grained enough to offer credible use in cryptanalysis then $\texttt{Dinur}_2$ is to be prefered.

12.2 Time complexities

In Table 9 below we see the time complexities for the algorithms not ruled out by the argumentation in the subchapter above. One should bear in mind, the differences in complexity measures, and thus any comparisons based directly on Table 9 must be done with care.

Algorithm [paper]	Complexity estimate	Expected or worst-case
`FES` [Bou+10]	$O(d\log_2(n)2^n)$	Expected
`Hybrid-`F_5 [BFP09]	$O(2^k m^\omega \binom{n-k-1+D_{reg}(n-k)}{D_{reg}(n-k)}^\omega)$	Expected
`Crossbred` [JV18]	See equation (56) below	Expected
$\texttt{Dinur}_1$[Din]	$O^*(2^{(1-\frac{1}{2d})n})$	Worst-case
$\texttt{Dinur}_2$[Din21]	$O(n^2 2^{(1-\frac{1}{2.7d})n})$	Expected

Table 9: PoSSo complexities

$$\tilde{O}\left(\left(\sum_{d_k=D_{kMac}+1}^{D}\sum_{d'=0}^{D-d_k}\binom{n-k_{inner}}{d_k}\binom{k_{inner}}{d'}\right)^2\right)+2^{k_{inner}}\tilde{O}\left(\left(\sum_{i=0}^{D_{kMac}}\binom{n-k_{inner}}{i}\right)^\omega\right) \quad (56)$$

12.3 Space complexities

The space complexities are less studied than the time complexities and are, in many cases, not analysed in any considerable detail. Therefore we are left to make comparisons about memory efficiency through arguments based on rough estimates on the space requirements of the algorithms. The space complexities for the five remaining algorithms to be compared are seen in Table 10 below.

Algorithm [paper]	**Space complexity**
`FES` [Bou+10]	*Negligible*
`Hybrid-`F_5 [BFP09]	*Unknown*
`Crossbred` [JV18]	*Unknown*
$Dinur_1$[Din]	$O^*(2^{(1-\frac{1}{2d})n})$
$Dinur_2$[Din21]	$O(n^2 2^{0.63n})$

Table 10: PoSSo$_2$ space complexities.

12.4 All-round comparisons

The merits of `FES`
Since the whole point of exhaustive search is to focus on avoiding overhead costs and hidden constants, it comes to the surprise of noone that `FES` must be viewed as the most efficient algorithm for *small problem instances*, i.e. for small n. This status is both due to the presumed small hidden constants in the time complexity and due to the low space demands.

Hybrid-F_5 vs `Crossbred`
Whilst the known time complexity estimates for `Hybrid-`F_5 and `Crossbred` look quite different, there are indications that in practice they behave similarily (see for example [Dua20]). If one considers the problem instances where these two algorithms perform the best, then it becomes clear that they occupy the same corner of the state-of-the-art. That is, they are *possibly* the fastest algorithms for *semi-regular, overdetermined systems with low degrees of regularity*. That this is the optimal scenario also for `Crossbred` stems from the fact that whilst the parameters in its complexity estimate do not directly involve the degree of regularity, they do directly consider the sizes of the involved matrices and clearly exploit dependence between the polynomials in the system. These attributes are also what the degree of regularity describes, namely, it is an indicator of how much the polynomials in the system relate to each other.

Neither of these two algorithms have a well understood space complexity. There are, however, promising experimental indications about their space requirements in that they are capable of solving comparably large problem instances in practice. For example was `Crossbred` able to solve a quadratic system with $n = 74, m = 148$ as a part of the Fukuoka challenge [Yas+15] .

The merits of Dinur's algorithms
The most obvious merit of Dinur's two algorithms is the low asymptotic time complexities that is only dependent on n and d and this merit directly suggests that one of these algorithms is the prime choice for *large systems, at least for problem instances poorly suited for* F_5 *and* `Crossbred`. The comparison between `Dinur`$_1$ and `Dinur`$_2$ mostly boils down to the intended use-case. On the one hand an advantage of `Dinur`$_1$ is that its complexity estimate is in the worst-case and that it has an asymptotically lower runtime. On the other hand, the hidden factors in its runtime are believed to be large. We argue that the randomness assumptions made in the analysis of `Dinur`$_2$ are highly defendable from the perspective of cryptanalysis. Further, the fact that the complexity estimate for the first algorithm ignores polynomial factors should be seen as such a blow to the credibility of its use in cryptanalysis

that for all intents and purposes, Dinur's second algorithm is to be preferred.

A major drawback of both of Dinur's algorithms is the exponential space complexity. This disadvantage must be seen as highly problematic since it indicates that implementing the algorithms and actually running them for large problem instances might be very challenging. On the other hand, one should bear in mind that the parameter choices that lead to the given time and space complexities are optimised with respect to time complexity and that both the parameter choices and details of the algorithms may, perhaps, be changed to arrive at more acceptable memory requirements.

For semi-regular systems
Deciding which algorithm is the fastest for any given problem instance is not a straight-forward task. This is partly due to the specific requirements of the entity solving the problem but it is partly due to the fact that it is not as easy as simply saying that `Hybrid-`F_5 is faster than $\texttt{Dinur}_2$ if the system is semi-regular. This becomes clear simply from looking at the formulas for the runtimes of the two algorithms, since which one of the algorithms is faster depends not only on n but also on d and the degree of regularity, amongst other factors. Therefore, one must unavoidably consider a few methods to see which one is likely to be the most efficient for a given problem instance.

13 Conclusion and discussion

Recall that the question posed in the introduction as the one that should be answered is

Q1: *What is the fastest method for solving a polynomial system over* $\mathbb{F}_2$*?*

The short answer that we propose is

A1: *In the general case and for large* n *then the newly introduced* `Dinur`$_2$ *algorithm is likely the fastest, but it does come at cost of exponential space requirements. However, if the system is semi-regular and has a low degree of regularity then* `Crossbred`*,* `Hybrid`*-*F_5 *or a similar algorithm is likely more efficient. For small systems,* `FES` *remains unsurpassed and similarily if one has strict restrictions on memory size.*

Below in Figure 10 follows a flowchart that may serve as a visualisation of A1. The flowchart may also serve as a basis for exploring which algorithms might be most effective for a given problem instance. Note, however, that it is not at all to be seen as exhaustive, since finding what algorithm is most efficient for a given system must necessarily consist of considering multiple different algorithms due to the large impact of system properties for algorithm performance.

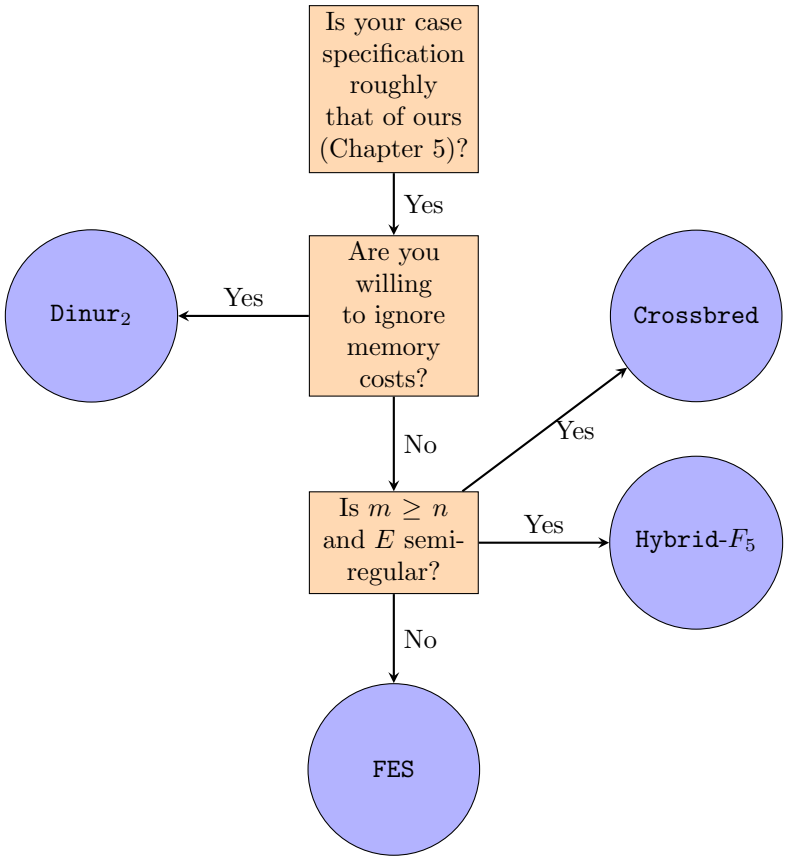

Figure 10: Flowchart over which algorithms might be most efficient for solving PoSSo$_2$.

The full answer is of course much more nuanced than A1 and therefore the rest of this chapter may be seen as a suitable extension of A1.

13.1 Flaws in our analysis

As always when one is offering a concise and simplified answer to nuanced questions, there are also regarding our analysis and conclusions many meritous arguments questioning our findings and our way to them. We will therefore now shortly consider some of them.

Dinur's second algorithm can not be considered state-of-the-art since it has no well tested implementation and because its space complexity is exponential.

The lack of a full implementation of the algorithm is primarily worrisome because without empirical evidence it is hard to argue for that the hidden factors (and terms) in the time complexity are in fact small enough to be omitted during analysis of systems of finite size. We would however claim that the lack of an implementation is not that problematic in the overall judgement of the algorithm's merits, except for that it further brings wood to the fire of questioning the implementability of the algorithm due to its space requirements. This point is probably the largest disadvantage of the algorithm, since one could argue that the algorithm could only be considered state-of-the-art after future results decrease the large memory demands. Whilst this line of arguing carries vast merit, we would rather mean that since the cryptanalytical use of these algorithms is inherently "over-theoretical" and reliant on narrow, conservative assumptions about future developments then one should reasonably consider $\texttt{Dinur}_2$ the most credible threat for solving system that are currently out of reach for it due to space requirements. This view is also supported by that the threshold of n where the algorithm likely requires less than 2^n bit operations is somewhat low (around $n = 65$ according to [BDT21]) and that one therefore should reason that relatively small improvements either in the algorithm (or analysis of it) or hardware could possibly bring the use of $\texttt{Dinur}_2$ on systems sizes where it should outperform `FES` from infeasability to feasability.

Passing judgement on the algorithms without thorough experimental comparisons inherently makes the conclusions unsuitably speculative.

It may ve tempting to make this objection since making comparisons of experimental results might feel less subjective and that since the entire value of the algorithms is the expectation that they are, or will in the future be, able to actually solve real instances of PoSSo. The merit of the argument that one should increase the scientific soundness of one's comparisons through comparing experimental results rather than through analytical arguments must however be seen as limited. This is due to the realisation that setting up such experiments necessarily introduce biases so to the degree that arguing that the experiments fundamentally reflect the merits of the algorithms is incredibly hard. One must, for example, motivate the choices of parameters of the systems, choice of hardware, choice of programming language and argue for how close a specific implementation is to an optimal implementation of each algorithm. Thus one should conclude that a lack of experimental comparisons, although they are doubtless a great resource if achieved properly, does not make the analysis overly speculative and this is since the experimental results must also be seen as comparably subjective to the argumentation (and mathematical results) that we have relied upon.

Comparing complexity results in different computational models is unstringent to the degree that conclusions cannot be credibly drawn without either experimental support or mathematical proofs of the relationships between the models or results.

This critisism, whilst harsh, is one that we to a large degree ourselves support and it was in fact a goal for us for a while during this project to provide such unifying proofs (or rather translations of complexity results into one single model). That goal was however abandoned, partly due to a change in our point of view but also largely due to the realisation that providing credible such translations would require vastly more time than was afforded to us in this project. The change in mind stems from the reasoning that all of the models we consider in this thesis are polynomial-time translatable to each other, meaning that the largest possible change in complexity from changing from one of the models to another is polynomial (we do, for example, know that the `TM` and the `RAM` have translation costs of at most N^2, where N is the size of the input [AHU74]). This does not mean that polynomial changes in complexities are to be lightly disregarded (see the discussion about the choice of big-O notation in Chapter 4.6) but it does mean that the comparative merits that resulted in the conclusion about the state-of-the-art for different problem instances are mostly indifferent to these changes. Therefore we landed in that the essence of the conclusions we draw are not critically affected by the inconsistency of computational model choices.

13.2 Unifying computational models

Even though we do not consider our conclusions invalidated by the fact that the algorithms use different models of computation, we do still claim that the inconsistency is gravely problematic for more concrete comparisons between pairs of algorithms. This is due to the fact that the use of the complexity estimates in cryptanalysis necessarily is coupled with arguments about the envisioned capabilities and goals of an attacker. Therefore one can not credibly use the complexity claims without arguments about how the computational model used to derive them correspond to the computational capacities of the hypothetical attacker.

For this reason, and since we mean that a complexity claim without a properly declared computational model should not be considered a mathematical proof but rather a particularily formalistic heauristic argumentation, the translation of complexity claims into new models of computation should still be seen as highly valuable. This value comes partly from the usefulness in actual comparisons but also, and perhaps primarily, from highlighting that stringently done complexity estimates must be a part of a satisfying analysis of an algorithm. Also, due to the role of cryptography as the basis of information security in general, we mean that the haphazard way complexity analyses at the moment are being done should be seen as a significant hole in the rigor of cryptography and thus problematic for a digitalised society at large.

As noted in the previous subchapter, our time restrictions hindered us from providing such proof translations and this was partly due to a percieved unfeasibly steep learning curve into writing complexity proofs and partly due to troubles in figuring out what is actually meant and assumed in currently available "proofs". Since almost all complexity claims reviewed are done without specifying a model of computation, a necessary first step to arriving at a proof in some by us predefined model must be to decipher the implicit assumptions made in the paper providing the results, a procedure we unfortunately deemed too time-demanding to be performed during this project.

13.3 Differences between $\mathbb{F}_2$ and larger fields

Most of the methods considered in this thesis only work, or only works with considerable merit, over the field $\mathbb{F}_2$ (for example `FES` and $\texttt{Dinur}_2$) and thus the landscape of the state-of-the-art for methods solving polynomial systems over other finite fields looks entirely different. As far as we are aware, for larger prime fields the cryptanalyst is left the options of either using a Gröbner basis algorithm or some version of the `XL` algorithm. The underlying reasons why some methods only work over the

binary field are quite interesting and they tell us much about the nuances and characteristics of both the fields themselves and the algorithms. Even more philosophically, one might ask if it is *inherently* so that the binary field is an "easier" domain to work in than, say, other prime fields or if it is simply that the state-of-the-art has come further for this case than it has for others.

A case for that the small size of the finite field directly implies that it allows for more efficient algorithms could be built on that the contrary indeed does feel unintuitive. This is partly due to the fact that in that case then the algorithm would somehow have to exploit the larger domain, which might seem hard, and partly due to that some of the properties that make $\mathbb{F}_2$ particularily convenient to work over has direct translations to similar properties in larger fields, with the caveat that they now are much less impactful.

Consider firstly *the field equations*, $x^p = x$, for the field $\mathbb{F}_p$, p prime. For the binary field this implies that polynomials cannot contain any variables with exponents larger than 1. The change of the field equations also means that, if one accepts the very sweeping statement that working with higher degree polynomials is in itself much slower than dealing with those of a lower degree, then utilising the field equations to, for example, guarantee specific algebraic properties or simplify the polynomial representation in a program quickly become unfeasible for fields of larger size. One concrete consequence of these less appealing field equations is that the construction of identifying polynomials used in the polynomial method becomes in practice unusable due to an exploding degree.

Why the arguments above should not lead to the conclusion that $\mathbb{F}_2$ is an inherently faster domain to solve polynomials over could be built partly on the idea that just because the direct translation of a construction in $\mathbb{F}_2$ does not work in a larger field then that does not necessarily have to mean that there is no good translation of said construction. A more compelling argument for this stance, perhaps, could be that in order to compare the inherent "efficiency of working over a field" then one needs to have a more nuanced idea of how one should measure the input size of a problem. One would therefore say that, for example, a problem size of n binary variables shouldn't be compared with a problem size of n variables in $\mathbb{F}_p$, but perhaps rather with a problem of, say, $\frac{n}{p}$ or $n^{\frac{1}{p}}$ variables.

13.4 Ethics

The topic of this thesis is to be seen as primarily belonging to the field of cryptology, and more specifically, cryptanalysis. The nature of this field in itself births the need to remark that its capability to undermine the security, in all its aspects, of vital technological systems carries with it a duty to wield these capabilities in a way that is beneficial to society at large. For this thesis specifically, one should remark that whilst it does not necessarily directly introduce any new offensive capacities, it does aim to give the reader of the thesis a better grip and understanding of a specific corner of cryptanalysis. This is not entirely unproblematic as it could allow a cryptanalysist with malicious intent to further hone their craft. One should, however, reasonably conclude that the ethically sound consequence and intention of allowing designers of systems to better understand the strength of their underlying cryptographic building blocks does indeed likely outweigh the risk of the insights we provide falling into the hands of maliciously minded individuals or organisations.

A note should perhaps be that whilst this ethical standpoint of full transparancy does, at first glance, seem to follow directly from *Kerckhoff's principle* (see [Pet11] for an english commentary), or equivalently *Shannon's maxim* [Sha49], we do believe that blindly extending that principle too far outside its intended interpretation would be unwise. Thus rather than relying on one of these old, established principles, let us say that we motivate the ethical purity of us conducting this thesis (and releasing it for public access) by that its effect on the wellfare of the populus through affecting the security of

digital infrastructure must be seen as neglible, and the small impact it might have is most reasonably seen as positive.

If one prefers a moral philosophical point of view other than the currently dominant implicit form of utilitarianism, we would say that Kant would probably be happy with us since we have not treated anyone as only a mean but also always as an end in themselves [Kan85] and Seneca would likely rejoice in our quest since we have focused on intellectual enrichment and thus become more resilient to possible future hardships [Sen 6][HD15].

13.5 Outlooks

Let us now finish this thesis by remarking on some possible directions future works could explore.

Write the translations of complexity proofs
As noted earlier, we do think that writing such proofs would be a great contribution to the literature on the topic, not only because it vastly improves the credibility of the resulting conclusions but also because it could contribute to a change in attitude within the cryptological community, namely the change to more rigorous complexity proofs.

Implement the most important methods and benchmark
Another possible continuation of this thesis would be to implement some of the most relevant algorithms under comparable circumstances, that is, for example, in the same programming language, with the same hardware in mind and having performed similar amounts of optimisations. Going down this route would contribute not only through providing experimental data to complement the argumentation given in this thesis, but also through providing a detailed discussion about how one would have to set up such experiments to ensure the highest possible credibility of one's conclusions.

Further investigate relationships between highly related algorithms
As remarked upon briefly in Chapter 13 and more at length in Chapter 8, there is considerable room for investigations into the relationships between some of the algorithms that share large similarities. For example, one could explore whether optimisations to `XL` regarding the matrix reductions (such as using the block-Wiedemann algorithm in `WXL`) can be applied to F_4 or explore further how the complexities and parameters of `Hybrid-`F_5 and `Crossbred` relate.

Continue the implementation of $\texttt{Dinur}_2$
Finally one could opt to continue the implementation of Dinur's second algorithm. This would carry with it the obvious value of granting better understanding to an extremely promising new algorithm. Additionally would this implementation, if sufficiently efficient, possibly become useful for solving actual problems involving finding roots of polynomial systems.

References

[Zou13] Yi Zou. "Representing Boolean Functions Using Polynomials: More Can Offer Less". In: *Advances in Neural Networks – ISNN 2011*. July 2013. ISBN: 978-3-642-21110-2. DOI: 10.1007/978-3-642-21111-9_32.

[FY79] A.S. Fraenkel and Y. Yesha. "Complexity of problems in games, graphs and algebraic equations". In: *Discrete Applied Mathematics* 1.1 (1979), pp. 15–30. ISSN: 0166-218X. DOI: https://doi.org/10.1016/0166-218X(79)90012-X.

[Nisb] *Post-Quantum Cryptography PQC, Round 3 Submissions*. Feb. 2022. URL: https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[Pat96] Jacques Patarin. "Hidden Fields Equations (HFE) and Isomorphisms of Polynomials (IP): Two New Families of Asymmetric Algorithms". In: *Advances in Cryptology — EUROCRYPT '96*. Ed. by Ueli Maurer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 33–48. ISBN: 978-3-540-68339-1.

[KPG99] Aviad Kipnis, Jacques Patarin, and Louis Goubin. "Unbalanced Oil and Vinegar Signature Schemes". In: *Advances in Cryptology — EUROCRYPT '99*. Ed. by Jacques Stern. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 206–222. ISBN: 978-3-540-48910-8.

[Nisa] *Post-Quantum Cryptography PQC, Round 2 Submissions*. Feb. 2022. URL: https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[JV18] Antoine Joux and Vanessa Vitse. "A Crossbred Algorithm for Solving Boolean Polynomial Systems". In: *Number-Theoretic Methods in Cryptology*. Ed. by Jerzy Kaczorowski, Josef Pieprzyk, and Jacek Pomykała. Springer International Publishing, 2018, pp. 3–21. ISBN: 978-3-319-76620-1.

[Din21] Itai Dinur. *Cryptanalytic Applications of the Polynomial Method for Solving Multivariate Equation Systems over GF(2)*. Cryptology ePrint Archive, Report 2021/578. https://ia.cr/2021/578. 2021.

[Din] Itai Dinur. "Improved Algorithms for Solving Polynomial Systems over GF(2) by Multiple Parity-Counting". In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 2550–2564. DOI: 10.1137/1.9781611976465.151. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611976465.151. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611976465.151.

[Bou89] N. Bourbaki. *Algebra I*. Springer Berlin Heidelberg, 1989. ISBN: 978-3-540-64243-5.

[Mor03] Teo Mora. *Solving Polynomial Equation Systems I: The Kronecker-Duval Philosophy*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2003. DOI: 10.1017/CBO9780511542831.

[Jou09] Antoine Joux. *Algorithmic cryptanalysis*. Chapman and Hall/CRC, 2009. ISBN: 9781420070026.

[KKW16] Petteri Kaski, Jukka Kohonen, and Thomas Westerbäck. "Fast Möbius Inversion in Semimodular Lattices and ER-labelable Posets". In: *Electron. J. Comb.* 23 (2016), P3.26.

[Bar+21] Stefano Barbero, Emanuele Bellini, Carlo Sanna, and Javier Verbel. *Practical complexities of probabilistic algorithms for solving Boolean polynomial systems*. Cryptology ePrint Archive, Report 2021/913. https://ia.cr/2021/913. 2021.

[BKW19] Andreas Björklund, Petteri Kaski, and Ryan Williams. "Solving Systems of Polynomial Equations over GF(2) by a Parity-Counting Self-Reduction". In: *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Ed. by Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi. Vol. 132. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 26:1–26:13. ISBN: 978-3-95977-109-2. DOI: 10.4230/LIPIcs.ICALP.2019.26.

[VV86] L.G. Valiant and V.V. Vazirani. "NP is as easy as detecting unique solutions". In: *Theoretical Computer Science* 47 (1986), pp. 85–93. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(86)90135-0.

[CLO07] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties and Algorithms*. Springer New York, NY, 2007. DOI: https://doi.org/10.1007/978-0-387-35651-8.

[Buc06] Bruno 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". In: *J. Symb. Comput.* 41 (2006), pp. 475–511. DOI: 10.1016/j.jsc.2005.09.007.

[Fau+93] J.C. Faugère, P. Gianni, D. Lazard, and T. Mora. "Efficient Computation of Zero-dimensional Gröbner Bases by Change of Ordering". In: *Journal of Symbolic Computation* 16.4 (1993), pp. 329–344. ISSN: 0747-7171. DOI: https://doi.org/10.1006/jsco.1993.1051.

[Mac16] F.S. Macaulay. *The algebraic theory of modular systems*. Cambridge University Press, 1916. ISBN: 1429704411.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. 2009. ISBN: 978-0-521-42426-4.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading (MA), 1974.

[Bou22] Charles Bouillaguet. *Nice Attacks — but What is the Cost? Computational Models for Cryptanalysis*. Cryptology ePrint Archive, Report 2022/197. 2022. URL: https://ia.cr/2022/197.

[Sav97] John E. Savage. *Models of Computation: Exploring the Power of Computing*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201895390.

[Tur37] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: https://doi.org/10.1112/plms/s2-42.1.230.

[CR73] Stephen A. Cook and Robert A. Reckhow. "Time bounded random access machines". In: *Journal of Computer and System Sciences* 7.4 (1973), pp. 354–375. DOI: https://doi.org/10.1016/S0022-0000(73)80029-7.

[Bac04] Paul Bachmann. "Analytische Zahlentheorie". In: *Encyklopädie der Mathematischen Wissenschaften mit Einschluss ihrer Anwendungen: Arithmetik und Algebra*. Ed. by Wilhelm Franz Meyer. B.G. Teubner Leipzig, 1904. ISBN: 978-3-663-16019-9. URL: https://doi.org/10.1007/978-3-663-16019-9_3.

[Lan09] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. B.G. Teubner Leipzig Berlin, 1909.

[IP01] Russell Impagliazzo and Ramamohan Paturi. "On the Complexity of k-SAT". In: *Journal of Computer and System Sciences* 62.2 (2001), pp. 367–375. ISSN: 0022-0000. DOI: https://doi.org/10.1006/jcss.2000.1727.

[Die11b] Claus Diem. "On the notion of bit complexity". In: *Bulletin of the European Association for Theoretical Computer Science EATCS* (2011).

[Sch80] A. Schönhage. "Storage Modification Machines". In: *SIAM Journal on Computing* 9.3 (1980), pp. 490–508. URL: https://doi.org/10.1137/0209036.

[Bou+10] Charles Bouillaguet et al. *Fast Exhaustive Search for Polynomial Systems in F_2*. Cryptology ePrint Archive, Report 2010/313. https://ia.cr/2010/313. 2010.

[Fau+17] Jean-Charles Faugère et al. *Fast Quantum Algorithm for Solving Multivariate Quadratic Equations*. Cryptology ePrint Archive, Report 2017/1236. 2017.

[Cou+00] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Ad Shamir. "Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations". In: *Advances in Cryptology — EUROCRYPT 2000*. Ed. by Bart Preneel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 392–407. ISBN: 978-3-540-45539-4.

[KS99] Aviad Kipnis and Adi Shamir. "Cryptanalysis of the HFE Public Key Cryptosystem by Relinearization". In: *Advances in Cryptology — CRYPTO' 99*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 19–30. ISBN: 978-3-540-48405-9.

[Che+14] Chen-Mou Cheng, Yasufumi Hashimoto, Hiroyuki Miura, and Tsuyoshi Takagi. "A Polynomial-Time Algorithm for Solving a Class of Underdetermined Multivariate Quadratic Equations over Fields of Odd Characteristics". In: *Post-Quantum Cryptography*. Ed. by Michele Mosca. Cham: Springer International Publishing, 2014, pp. 40–58. ISBN: 978-3-319-11659-4.

[Fau99] Jean-Charles Faugére. "A new efficient algorithm for computing Gröbner bases (F4)". In: *Journal of Pure and Applied Algebra* 139.1 (1999), pp. 61–88. DOI: https://doi.org/10.1016/S0022-4049(99)00005-5.

[Fau02] Jean Charles Faugére. "A New Efficient Algorithm for Computing GröBner Bases without Reduction to Zero (F5)". In: ISSAC '02. Lille, France: Association for Computing Machinery, 2002, 75–83. ISBN: 1581134843. DOI: 10.1145/780506.780516.

[BFP09] Luk Bettale, Jean charles Faugère, and Ludovic Perret. "Hybrid approach for solving multivariate systems over finite fields". In: *JOURNAL OF MATHEMATICAL CRYPTOLOGY* (2009), pp. 177–197.

[Din+08] Jintai Ding et al. "Mutantxl". In: *SCC* (Jan. 2008), pp. 16–22.

[Bar+13] Magali Bardet, Jean-Charles Faugère, Bruno Salvy, and Pierre-Jean Spaenlehauer. "On the complexity of solving quadratic Boolean systems". In: *Journal of Complexity* 29.1 (2013), pp. 53–75. DOI: https://doi.org/10.1016/j.jco.2012.07.001.

[BDT21] Charles Bouillaguet, Claire Delaplace, and Monika Trimoska. *A Simple Deterministic Algorithm for Systems of Quadratic Polynomials over* $\mathbb{F}_2$. Cryptology ePrint Archive, Report 2021/1639. 2021. URL: https://ia.cr/2021/1639.

[Lok+] Daniel Lokshtanov et al. "Beating Brute Force for Systems of Polynomial Equations over Finite Fields". In: *Proceedings of the 2017 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 2190–2202. DOI: 10.1137/1.9781611974782.143. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611974782.143.

[Yas+15] Takanori Yasuda et al. *MQ Challenge: Hardness Evaluation of Solving Multivariate Quadratic Problems*. Cryptology ePrint Archive, Report 2015/275. https://ia.cr/2015/275. 2015.

[Gra47] Frank Gray. *Pulse Code Communication*. U.S. Patent 2,632,058. 1947.

[Ars+04] Gwénolé Ars et al. "Comparison Between XL and Gröbner Basis Algorithms". In: *Advances in Cryptology - ASIACRYPT 2004*. Ed. by Pil Joong Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 338–353. ISBN: 978-3-540-30539-2.

[BFS03] Magali Bardet, Jean-Charles Faugére, and Bruno Salvy. *Complexity of Gröbner basis computation for Semi-regular Overdetermined sequences over* F_2 *with solutions in* F_2. Research Report RR-5049. INRIA, 2003. URL: https://hal.inria.fr/inria-00071534.

[Nak+20] Shuhei Nakamura et al. *New Complexity Estimation on the Rainbow-Band-Separation Attack*. Cryptology ePrint Archive, Paper 2020/703. https://eprint.iacr.org/2020/703. 2020. URL: https://eprint.iacr.org/2020/703.

[CGH91] L. Caniglia, A. Galligo, and J. Heintz. "Equations for the projective closure and effective Nullstellensatz". In: *Discrete Applied Mathematics* 33.1 (1991), pp. 11–23. ISSN: 0166-218X. DOI: https://doi.org/10.1016/0166-218X(91)90105-6. URL: https://www.sciencedirect.com/science/article/pii/0166218X91901056.

[CP03] Nicolas T. Courtois and Jacques Patarin. "About the XL Algorithm over GF(2)". In: *Topics in Cryptology — CT-RSA 2003*. Ed. by Marc Joye. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 141–157. ISBN: 978-3-540-36563-1.

[TW10] Enrico Thomae and Christopher Wolf. *Solving Systems of Multivariate Quadratic Equations over Finite Fields or: From Relinearization to MutantXL*. Cryptology ePrint Archive, Paper 2010/596. https://eprint.iacr.org/2010/596. 2010. URL: https://eprint.iacr.org/2010/596.

[AM11] Wael Said Abdelmageed Mohamed. "Improvements for the XL Algorithm with Applications to Algebraic Cryptanalysis." 2011. URL: https://tuprints.ulb.tu-darmstadt.de/id/eprint/2621.

[YC04] Bo-Yin Yang and Jiun-Ming Chen. "All in the XL Family: Theory and Practice". In: *Lecture Notes in Computer Science*. Vol. 3506. Dec. 2004, pp. 67–86. ISBN: 978-3-540-26226-8. DOI: 10.1007/11496618_7.

[Moh+10] Wael Said Abd Elmageed Mohamed et al. "PWXL: A Parallel Wiedemann-XL Algorithm for Solving Polynomial Equations over GF(2)". In: *International Conference on Symbolic Computation and Cryptography – SCC*. Ed. by Carlos Cid and Jean-Charles Faugére. 2010, pp. 89–100.

[Wie86] D. Wiedemann. "Solving sparse linear equations over finite fields". In: *IEEE Transactions on Information Theory* 32.1 (1986), pp. 54–62. DOI: 10.1109/TIT.1986.1057137.

[FK21] Hiroki Furue and Momonari Kudo. *Polynomial XL: A Variant of the XL Algorithm Using Macaulay Matrices over Polynomial Rings*. Cryptology ePrint Archive, Report 2021/1609. 2021. URL: https://ia.cr/2021/1609.

[Alb+12] Martin Albrecht, Carlos Cid, Jean-Charles Faugére, and Ludovic Perret. "On the relation between the MXL family of algorithms and Gröbner basis algorithms". In: *Journal of Symbolic Computation* 47 (2012). DOI: 10.1016/j.jsc.2012.01.002.

[Che+16] Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, and Bo-Yin Yang. *Solving Quadratic Equations with XL on Parallel Architectures - extended version*. Cryptology ePrint Archive, Report 2016/412. 2016. URL: https://ia.cr/2016/412.

[Dua20] João Diogo Duarte. *On the Complexity of the Crossbred Algorithm*. Cryptology ePrint Archive, Report 2020/1058. https://ia.cr/2020/1058. 2020.

[NNY18] Ruben Niederhagen, Kai-Chun Ning, and Bo-Yin Yang. "Implementing Joux-Vitse's Crossbred Algorithm for Solving MQ Systems over $\mathbb{F}_2$ on GPUs". In: *Post-Quantum Cryptography*. Ed. by Tanja Lange and Rainer Steinwandt. Cham: Springer International Publishing, 2018, pp. 121–141. ISBN: 978-3-319-79063-3.

[Wil14] Richard Williams. "The Polynomial Method in Circuit Complexity Applied to Algorithm Design". In: *34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014)*. Vol. 29. 2014, pp. 47–60. DOI: 10.4230/LIPIcs.FSTTCS.2014.47.

[Raz87] Aleksandr Aleksandrovich Razborov. "Lower bounds on the size of bounded depth circuits over a complete basis with logical addition". In: *Mathematical Notes of the Academy of Sciences of the USSR 41*. 1987, pp. 333–338. DOI: https://doi.org/10.1007/BF01137685.

[Smo87] Roman Smolensky. "Algebraic methods in the theory of lower bounds for boolean circuit complexity". In: *IN PROCEEDINGS OF THE 19TH ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING, STOC '87*. 1987, pp. 77–82.

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Pet11] Fabien A. P. Petitcolas. "Kerckhoffs' Principle". In: *Encyclopedia of Cryptography and Security*. Ed. by van Tilborg, Henk C. A., and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 675–675. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_487. URL: https://doi.org/10.1007/978-1-4419-5906-5_487.

[Sha49] C. E. Shannon. "Communication theory of secrecy systems". In: *The Bell System Technical Journal* 28.4 (1949), pp. 656–715. DOI: 10.1002/j.1538-7305.1949.tb00928.x.

[Kan85] Immanuel Kant. *Grundlegung zur Metaphysik der Sitten*. J.F. Hartknoch, 1785.

[Sen 6] Lucius Annaeus Seneca. *Epistulae morales ad Lucilium*. ca 64 BC.

[HD15] Andreas Heil and Gregor Damschen. *Brill's Companion to Seneca: Philosopher and Dramatist*. Leiden, The Netherlands: Brill, 2015. ISBN: 978-90-04-21708-9. DOI: https://doi.org/10.1163/9789004217089. URL: https://brill.com/view/title/12087.

[Die11a] Claus Diem. *On the complexity of some computational problems in the Turing model*. 2011. URL: https://www.math.uni-leipzig.de/~diem/preprints/turing.pdf.

[Sto00] Arne Storjohann. "Algorithms for matrix canonical forms". 2000. URL: https://doi.org/10.3929/ethz-a-00414100.

[Bou+13] Charles Bouillaguet et al. *Fast Exhaustive Search for Quadratic Systems in $\mathbb{F}_2$ on FPGAs — Extended Version*. Cryptology ePrint Archive, Report 2013/436. 2013. URL: https://ia.cr/2013/436.

[BCJ07] Gregory V. Bard, Nicolas T. Courtois, and Chris 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. URL: https://ia.cr/2007/024.

[Bar09] Gregory Bard. *Algebraic cryptanalysis*. Springer Dordrecht Heidelberg London New York, 2009. ISBN: 9780387887562.

[BFS04] Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. *On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations*. 2004. URL: https://www-polsys.lip6.fr/~jcf/Papers/43BF.pdf.

A Problem version definitions

Definition A.1 (The PoSSo$_2$-Decision problem)**.**
Input: *A polynomial system $E \subset \mathbb{F}_2[\mathbf{x}]$.*
Output: *One bit describing whether or not there exists a root of E. 0 means there is no root and 1 means there is at least one.*

Definition A.2 (The PoSSo$_2$-Exhaustive problem)**.**
Input: *A polynomial system $E \subset \mathbb{F}_2[\mathbf{x}]$.*
Output: *A list of all roots of E.*

Definition A.3 (The PoSSo$_2$-Count problem)**.**
Input: *A polynomial system $E \subset \mathbb{F}_2[\mathbf{x}]$.*
Output: *An integer describing the number of roots to E.*

Definition A.4 (The PoSSo$_2$-ParityCount problem)**.**
Input: *A polynomial system $E \subset \mathbb{F}_2[\mathbf{x}]$.*
Output: *One bit describing whether the number of roots to E is even or odd. 0 means even and 1 means odd.*

Definition A.5 (The PoSSo$_2$-MultParityCount problem)**.**
Input: *A polynomial system $E \subset \mathbb{F}_2[\mathbf{x}]$ and two non-negative integers $k \leq n$ and $w \leq n-k$.*
Output: *A binary vector $V \in \mathbb{F}_2^{\binom{n-k}{\downarrow w}}$ such that the i:th value of V is the parity of the number of roots where the first $n-k$ variables are assigned to the i:th value in W_w^{n-k}.*

B Problem version reductions

The reductions between problem versions that are of interest to us are those that leads to the result that PoSSo$_2$-Search reduces to each other problem version in polynomial time. First of all we note that a solution to PoSSo$_2$-Exhaustive trivially yields a solution to PoSSo$_2$-Search in constant time and similarily does solving PoSSo$_2$-Count trivially solve PoSSo$_2$-Decision in constant time. What then remains to show is that PoSSo$_2$-Search reduces to PoSSo$_2$-Decision, Decision to ParityCount and ParityCount to MultParityCount, all in at most polynomial time.

Proposition B.1 (PoSSo$_2$-Search reduces to PoSSo$_2$-Decision (Reformulation of Chapter 2.4 in [BKW19]))**.**
Given an algorithm, call it $\mathcal{A}$, that solves PoSSo$_2$-Decision, there is an algorithm that solves PoSSo$_2$-Search using at most $2n$ executions of $\mathcal{A}$.

Proof-sketch (Reformulation of Chapter 2.4 in [BKW19]):
For each x_i, assign it to 0 and 1 respectively and evaluate if there is a solution for that specific assigment. If there is, then fix x_i at that assignment and move on to another variable. If there is no satisfying assignment for a variable, then output that there is no solution to the PoSSo$_2$-Search instance. Thus the algorithm will, if there is a solution to the PoSSo$_2$-Search instance give it from at most $2n$ evaluations of $\mathcal{A}$ and if there is no solution to the PoSSo$_2$-Search instance then the algorithm will determine that after at most 2 evaluations of $\mathcal{A}$. □

Proposition B.2 (PoSSo$_2$-Decision reduces to PoSSo$_2$-ParityCount (Reformulation of Chapters 2.4 and 2.5 in [BKW19])).
Given an algorithm, call it $\mathcal{A}$, that solves PoSSo$_2$-ParityCount then there is an algorithm that solves PoSSo$_2$-Decision with high probability in at most $O(nlog_2(n))$ queries to $\mathcal{A}$.

Proof-sketch(Reformulation of Chapters 2.4 and 2.5 in [BKW19]):
First note that if one knows that the polynomial system in question has at most one root, then the parity of the number of solutions directly gives the satisfiability of the system, since an odd parity can only mean that there is a root and an even parity can only mean that there is no root. Thus what remains to show is that a solution may always be isolated. Through utilising Valiant-Vazirani affine hashing (see Chapter 2.5), one can isolate a root to E with probability at least $1 - \frac{1}{n}$ (which is close to 1) by performing $\lceil log_2(n) \rceil$ independent repetitions of Valiant-Vazirani affine hashing. This is however only true under the assumption that one knows approximately the number of roots to E, which we do not. More precisely does one need to know the $c \in \{0, ..., n\}$ such that the number of roots lies in between 2^c and 2^{c+1}. Therefore this procedure needs to be repeated for each possible guess of c, and thus does $O(n \cdot log_2(n))$ queries to $\mathcal{A}$ lead to a solution of PoSSo$_2$-Decision *with high probability.* □

Proposition B.3 (PoSSo$_2$-MultParityCount to PoSSo$_2$-ParityCount (reformulation of Chapters 2.4 - 3.1 in [Din])).
Given an algorithm, call it $\mathcal{A}$, that solves PoSSo$_2$-MultParityCount then there is an algorithm that solves PoSSo$_2$-ParityCount in at most one query to $\mathcal{A}$.

Proof-sketch(reformulation of Chapters 2.4 - 3.1 in [Din]):
Remember from Chapter 10.3 that the parity of the number of solutions is precisely $\sum_{\mathbf{x}^* \in \mathbb{F}_2^n} F(\mathbf{x}^*)$, where F is the identifying polynomial of E. This sum may be split up according to a variable partition into

$$\sum_{\mathbf{x}^* \in \mathbb{F}_2^n} F(\mathbf{x}^*) = \sum_{\mathbf{y}^* \in \mathbb{F}_2^{n-k}} \sum_{\mathbf{z}^* \in \mathbb{F}_2^k} F(\mathbf{y}^*, \mathbf{z}^*) = \sum_{\mathbf{y}^* \in \mathbb{F}_2^{n-k}} G(\mathbf{y}^*).$$

Thus, due to the fact that we may interpolate $G(\mathbf{y}^*)$ from the evaluations of it on W_{dl-k}^{n-k} (defined as in Chapter 2.4), then PoSSo$_2$-ParityCount may be solved in at most one query to $\mathcal{A}$, since $\mathcal{A}$ returns precisely such evaluations of $G(\mathbf{y}^*)$. □