Boolean Polynomial Evaluation for the Masses

Charles Bouillaguet

LIP6 laboratory, Sorbonne Université, Paris, France
charles.bouillaguet@lip6.fr

Abstract. This article gives improved algorithms to evaluate a multivariate Boolean polynomial over all the possible values of its input variables. Such a procedure is often used in cryptographic attacks against symmetric schemes. More precisely, we provide improved and simplified versions of the Fast Exhaustive Search algorithm presented at CHES'10 and of the space-efficient Moebius transform given by Dinur at EUROCRYPT'21. The new algorithms require $\mathcal{O}(d2^n)$ operations with a degree-d polynomial and operate in-place. We provide the full C code of a complete implementation under the form of a "user-friendly" library called BeanPolE, which we hope could be helpful to other cryptographers. This paper actually contains all the code, which is quite short.

Keywords: Boolean polynomials · exhaustive search · Moebius transform · software implementation

When in doubt, use brute force.

Attributed to Ken Thompson [Ja22], co-inventor of Unix

1 Introduction

We consider the natural problem of efficiently evaluating a Boolean polynomial (given by its coefficients) on all possible inputs. This provides a way to build its truth table. Any Boolean function $f(x_0, \ldots, x_{n-1})$ on n variables can be completely described by providing its complete truth table, namely the 2^n bits that give its value on each of the possible values of the input variables.

Low-degree Boolean polynomials are Boolean function that admit a much more compact representation. For instance, a Boolean quadratic polynomial

$$f(x_0, \ldots, x_{n-1}) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} a_{ij} x_i x_j + \sum_{i=0}^{n-1} b_i x_i + c$$

is entirely described by the values of its $n(n+1)/2+1$ coefficients. In general, a degree-d Boolean polynomial has $\mathcal{O}\left(n^d\right)$ coefficients. As such, Boolean polynomial are easier to manipulate than generic Boolean functions, even with a high number of variables.

We describe two procedures to *visit* all entries of the truth table of a degree-d polynomial f in n variables given by its coefficients. This offers a way to iterate over pairs $(x, f(x))$ for all bit strings $x \in \{0,1\}^n$. These two procedures are *in-place*: they require only a small number of words of extra memory in addition to the space needed to store the polynomial f.

The first procedure is a slight variation of the "Fast Exhaustive Search" (FES) algorithm of Bouillaguet, Chen, Cheng, Chou, Niederhagen, Shamir and Yang [BCC+10a]. The

second procedure is a slight variation of the "Space-Efficient Moebius Transform" given by Dinur [Din21a].

In the sequel, we will often omit the word "Boolean" ; polynomials, monomials, variables, etc. are all Boolean.

1.1 Solving Generic Polynomial Systems

A procedure that iterates over the truth table of a Boolean polynomial can be used to solve systems of polynomial equations by exhaustive search. This is an NP-hard problem even when restricted to quadratic polynomials (there is a straightforward reduction to 3SAT) [GJ79].

Besides the natural intellectual prospect that consists in inventing better algorithms for this simple problem, the initial motivation for developing efficient exhaustive search algorithms stems from cryptanalysis.

The so-called "Multivariate" public-key encryption and signatures schemes expose a public key which is a collection of m quadratic polynomials in n variables over a finite field $\mathbb{F}_q$. Solving this polynomial system breaks the security properties offered by the scheme. Some recent public-key signature schemes that fall in this category have been submitted to the "post-quantum" competition organized by the NIST [CHR+16, DS05, CFMR+17, BP17]. The designers of these schemes are faced with the problem of choosing the parameters (n, m, q) to make the scheme simultaneously secure and efficient. Developing and analyzing algorithms to solve polynomial systems therefore helps getting a better understanding of the hardness of the problem. To track practical progress, a collection of challenges has been made available online [YDH+15].

When it was developed in 2010, the FES algorithm was undisputedly the best solution to solve unstructured low-degree Boolean systems *in practice*. Existing asymptotically better algorithms such as hybrid variants of Faugere's F4 or F5 algorithms [Fau99, Fau02], along with their cousins from the XL family [CKPS00, YC04], could simply not compete in terms of speed. In addition, their exponential space complexity made them practically unusable.

Efficient implementations of the FES algorithm claimed several speed records. This culminated in 2015 with the resolution of a random 66-variable, 99-equation challenge[1] by Chou, Niederhagen and Yang using an FPGA implementation of the algorithm [BCC+13].

From a theoretical point of view, on the day it was published, the FES algorithm had the best asymptotic complexity among algorithms capable to solve arbitrary polynomial systems in a worst-case sense, without requiring the assumption that the polynomial system is "generic" enough — algebraic techniques that claimed a better complexity need this assumption. The FES algorithm lost the crown in 2017, when a better worst-case algorithm was invented by Lokshtanov, Paturi, Tamaki, Williams and Yu [LPT+17], based on the "polynomial method for algorithm design". This new algorithm and its recent derivatives [BKW19, Din21b, Din21a] are nevertheless highly impractical.

From a practical point of view, the situation also changed radically in 2017 with the advent of the Crossbred algorithm of Joux and Vitse [JV17]. It solved a random system of 148 equations in 74 variables[2], something that would not have been possible by exhaustive search. This was a stunning demonstration that "beating brute force" *in practice* was possible. In addition, Joux and Vitse estimated that their code (which is not public) was faster than the available implementation of the FES algorithm on systems with 40 variables or more, a very low and practical threshold.

This meant that developing or maintaining fast implementations of exhaustive search could no longer lead to either the fastest possible theoretical algorithms or the fastest practical software packages to solve Boolean polynomial systems.

[1] https://www.mqchallenge.org/details/details_IV_20150713.html
[2] https://www.mqchallenge.org/details/details_I_20161217_2.html

1.2 Algebraic Cryptanalysis

Algorithms to solve polynomial systems constitute the workhorse of "algebraic cryptanalysis". Some cryptographic schemes can be broken by writing a clever system of polynomial equations and solving it using off-the-shelf tools. This idea can be traced back to the work of Shannon [Sha49]. Any algorithm capable of solving polynomial systems could be used; exhaustive search can be a simple and reasonable solution.

In some cases, the polynomial systems can be solved in practice; this leads to practical attacks. This has been the case for the LUOV [BP17] signature scheme submitted to the recent "Post-quantum" public-key cryptography competition organized by the NIST. Signatures could be forged in less than 4 hours by solving two systems with 59 variables and 57 quadratic equations [DDVY21], using the aforementioned implementation of FES on FPGAs [BCC+13].

The next two examples are not strictly speaking about Boolean systems, but the results are nonetheless impressive.

The Rainbow signature scheme was also practically broken by Beulens [Beu22] during the competition, this time using an implementation of the Block Wiedemann XL solver by Cheng, Chou, Niederhagen, and Yang [CCNY12]. This solves a system of 63 quadratic equations in 30 variables over $\mathbb{F}_{16}$ in 3 hours and a half on a laptop.

In the context of symmetric cryptography, Bariant, Bouvier, Leurent and Perrin recently proposed algebraic attacks against "Arithmetization-Oriented" primitives [BBLP22]. They build univariate or multivariate polynomials over a large finite field and either solve them in practice or estimate the complexity of doing so to attack Feistel-MiMC, Poseidon, Rescue-Prime and Ciminion.

Most of the time, however, the systems cannot be solved in practice, and the attacks remain theoretical. In that case, the complexity of the solver has the greatest importance, because it determines whether the attack breaks the cryptographic scheme or not.

In the realm of symmetric cryptography, examples are provided by the recent attacks against the block-cipher LowMC due to Liu, Meier, Sarkar and Isobe [LMSI22] and also to Dinur [Din21a]. The former uses the complexity of an easy-to-analyze special case of the Crossbred algorithm [BDT22] to estimate the total time needed to complete the attack, while the latter proposes a new algorithm with a precise complexity analysis. A bit earlier, Duval, Lallemand and Rotella [DLR16] broke the FLIP stream cipher; their attack solves a multivariate quadratic Boolean system by linearization.

1.3 Actually Building the Truth Table

Some cryptographic attacks have an "algebraic phase" where the full truth table of a multivariate polynomial has to be computed.

For instance, Dinur and Shamir proposed an attack [DS11] against the hash function Hamsi-256, which crucially requires a fast procedure to evaluate degree-6 polynomials on all possible values of their 32 variables. More recently, the attack [BDL+21] of Beierle, Derbez, Leander, Leurent, Raddum, Rotella, Rupprecht, and Stennes against the GPRS Encryption Algorithms GEA-1 and GEA-2 (in "2G cellphones") repeatedly builds the truth table of several degree-4 polynomials in 33 variables. The currently best known key-recovery attack on Trivium is due to Hu, Sun, Todo, Wang and Wang [HST+21]. In its last phase, it builds the truth table of several (sparse) Boolean polynomials of degree about 20 in up to 75 variables. The attack against Pyjamask-96 proposed by Dobraunig, Rotella and Schoone [DRS20] repeatedly evaluate (potentially sparse) degree-4 Boolean polynomials on ≈ 128 variables.

In a very different context, the most computationally expensive phase of the Crossbred algorithm does the following: evaluate several degree-d polynomials on the next value

of the (many) input variables, then solve a linear system. As such, it actually iterates through the truth table of Boolean polynomials.

If a polynomial has n variables, then its truth table has 2^n entries, and this is necessarily a lower-bound on the time complexity of any algorithm that computes it. "Smart" algorithms are thus excluded, and techniques similar to exhaustive search rule.

The simplest option is to evaluate the polynomial naively on each possible input. This results in $\mathcal{O}\left(n^d 2^n\right)$ bit operations and works *memoryless* (*i.e.*, this does not require *any* additional memory besides what is needed to store the coefficients of the polynomial). The FES algorithm improves the time complexity to $\mathcal{O}\left(d2^n\right)$ operations.

Any Boolean function can be represented as a multivariate polynomial. The array of its coefficients is the Algebraic Normal Form of the function. A simple FFT-like algorithm, which we call the Moebius Transform [Jou09, §9.2] in this article, converts the truth table of a function to its algebraic normal form and vice-versa in $\mathcal{O}\left(n2^n\right)$ operations (the procedure is involutive). It works in-place, but it necessarily operates on an array of 2^n bits, because its output is the full truth table. As such, in practice it is restricted to fairly small values of n. It must be noted that this algorithm is easy to implement. A complete C implementation is given in [Jou09]; it fits in less than one page.

For high-degree polynomials (of degree greater than $n/2$), the Moebius transform is not much worse than enumeration algorithm such as FES. Indeed, it is well known that $\binom{n}{n/2} \sim 2^n/\sqrt{\pi n/2}$; this implies that a degree-$(n/2)$ polynomial has $\mathcal{O}\left(2^n/\sqrt{n}\right)$ monomials of degree $n/2$, and therefore storing it in memory almost requires as much space as storing an arbitrary Boolean function. As a consequence, "low-degree" polynomials are those with degree significantly less than $n/2$. For high-degree polynomials, the Moebius transform offers comparable, if not better performance than competing solutions.

Two improved Moebius transforms have been described as by-products of cryptographic attacks. Starting from the fact that the algebraic normal form of a degree-d polynomial has $\mathcal{O}\left(n^d\right)$ non-zero coefficients instead of 2^n in the worst case, Dinur and Shamir [DS11] designed an improved Moebius transform that skips useless operations with coefficients that are known to be zero. It requires only $\mathcal{O}\left(d2^n\right)$ operations but produces the full truth table, and thus requires 2^n bits of memory.

In [Din21a], Dinur described a space-efficient version which combines exhaustive search and the classic Moebius transform. The idea consists in enumerating all the possible values of k variables, then performing the Moebius transform on the resulting degree-d polynomial in $n-k$ variables. If k is chosen such that 2^{n-k} is close to the number of monomials of the input polynomial, then the algorithm only requires a linear amount of extra storage.

These two variants can in principle be combined.

1.4 A Survey of Previous Implementations of the FES algorithm

The FES algorithm enumerates all possible inputs by only flipping one variable at a time, using a Gray code. Updating the output of the polynomial function when a single variable is flipped requires the evaluation if its "derivative", which is a polynomial of degree-$(d-1)$. The crucial observation is that this derivative is itself evaluated on related points that only differ by two bits. The same idea can then be used recursively.

Previous implementations of the FES algorithm were developed with the objective of maximum speed, in particular for quadratic polynomials. As a result, these programs were complex and difficult to maintain.

When the article presenting the FES algorithm was published in 2010 [BCC+10a], only a "research-quality" implementation was available. This means that it could barely run on the authors own machines and was just good enough to produce (hard-to-reproduce) timings for a research publication. Only degree 2,3 and 4 were supported, by different

programs. A GPU version[3] and an FPGA version [BCC+13] have also been developed. To the best of our knowledge, the original CPU code has never been published; this shows how useful it was.

Between 2012 and 2014, Bouillaguet and Chou wrote a slightly better version dubbed libFES[4]. It was more usable, was capable of dealing with systems of degree less than a given compile-time bound, using a different function for each degree. It had bindings to the SageMath [The13] computer algebra system for a period of time. It has been abandoned because maintenance was too hard, and the SageMath bindings were therefore removed.

Starting from 2017, Bouillaguet wrote a simpler and smaller library called libfes-lite[5], restricted to *quadratic* Boolean systems. The underlying assumption was that this restriction would make development more manageable. The algorithm was streamlined and simplified. In terms of usability, libfes-lite provides a standalone multi-threaded program that reads a system from a text file in a simple format and print its solutions. It has an extensive test suite, which helps build confidence in its correctness. It is the fastest CPU-only implementation of exhaustive search for Boolean quadratic polynomials at the time of this writing. This comes at the expense of platform-specific bits of assembly code generated by *ad hoc* python scripts.

Unfortunately, because this last version is restricted to quadratic polynomials only, it is unusable in all the relevant cryptanalytic scenarios discussed in section 1.3.

1.5 Algorithmic Contributions

We describe two algorithms to iterate over the truth table of a Boolean polynomial of arbitrary degree. They both improve over the state of the art.

Our version of the FES algorithm improves upon the original presentation given in the extended version of [BCC+10a] as follows:

- It is much simpler and dispenses with the complex recursive and "object-oriented" structure of the original algorithm.
- It is in-place. It alters the polynomial, but restores it to its original state once the full truth table has been visited. It requires $\mathcal{O}(nd)$ extra words of storage (in addition to the space needed for the polynomial).
- It visits the next entry of the truth table in $\mathcal{O}(d)$ operations in the worst case. The original presentation does it in amortized constant-time only.
- The setup phase is simpler and comes with a better complexity analysis.

Obtaining the whole truth table of f by iterating the procedure 2^n times requires $\mathcal{O}(d2^n)$ operations.

Our version of the Moebius transform improves upon the presentation given in [Din21a] as follows:

- It is in-place, and in fact it does not require *any* other memory than what is needed to store the polynomial and the current values of the input variables.
- It runs in time $\mathcal{O}(d2^n)$ on usual computers, as opposed to the original version for which it is unclear (see Subsection 1.7).

[3] http://polycephaly.org/projects/forcemq/
[4] https://gitlab.lip6.fr/bouillaguet/old_fes
[5] https://gitlab.lip6.fr/almasty/libfes-lite

1.6 Literate Programs

Our algorithms are completely described in this article in the form of *literate programs* [Knu92] in the C programming language. A litterate program is addressed to human beings rather than to a computer. The `noweb` [Ram94] system can be used to extract the source code of (working) implementations from the source of this document. The interested reader will find the complete working code online at

`https://gitlab.lip6.fr/almasty/litFES`

This unusual style of presentation has the following features:

- The algorithms are described in a rather formal and a non-ambiguous way (in the C language).
- It leads to clean, understandable and well-documented code. This maximizes the potential reuse by third parties.
- It emphasizes the practicality of the algorithm.
- There are no interesting "implementations details" left to the reader.

The running time of individual statements is the C language is rather straightforward to analyze, except function calls. The `size_t` type essentially corresponds to the largest unsigned integer type natively supported by the hardware. This allows a C program to somehow emulate the standard and ubiquitous *transdichotomous* computational model in which most algorithms are described [FW93, FW94, CLRS09].

If a very precise model of computational complexity had to be explicitly given, then the number of instructions executed by a RISC-V processor running the compiled C code seems relevant. The algorithms given in this paper do not need multiplications, so the RVxxxI ISA is actually sufficient, where xxx denotes the width of registers.

In contrast to the previous versions of the FES algorithm that were speed demons, this new implementation targets flexibility, portability, simplicity, conciseness and ease of use. This design decision is motivated by the recent progress in algorithms for solving systems of Boolean quadratic equations described in section Subsection 1.1. Exhaustive search will no longer be the fastest available algorithm to solve polynomial systems. However it can still be a useful tool in cryptographic attacks that need to build the truth table of a polynomial, as discussed in section Subsection 1.3. The code presented here has been written with this objective in mind.

To maximize potential reuse, the algorithms are presented as a C library called BeanPolE (BoolEAN POLynomial Evaluation), with a simple and convenient user interface. We believe that providing a library, as opposed to a standalone "demonstration program", could make it easier for others to reuse the algorithms in their own software projects.

The BeanPolE library currently offers two dual components: litFES (a LITerate implementation of Fast Exhaustive Search) and litMOB (a LITerate in-place MOeBius transform), which are fully described in Section 6 and Section 7, respectively.

In addition to the library itself, we provide a few demonstration programs which exemplify its use. Separating the library from the demonstration programs allows this article to focus on the main algorithmic points (covered by the library), while leaving aside irrelevant and mundane details, such as parsing arbitrary degree polynomials from text files (covered by the demonstration programs).

1.7 The Devil Lies in the Details

> You are assuming that the algorithm is implemented on a processor [...]
>
> Itai Dinur, private communication

We advocate that providing a complete implementation of algorithms has benefits that may not be completely obvious at first sight.

When it is not the case, it is up to the readers to implement the algorithm themselves. This activity (turning the high-level description of an algorithm given in a research publication into an actual program using a common programming language) is sometimes perceived as a straightforward engineering task, as opposed to a research work that requires the invention of new algorithmic ideas[6].

But sometimes the situation is a bit more complicated than it should be. The first reason is that hidden and problematic "implementation details" may lie dormant in the high-level descriptions of some algorithms, and this in turn may make it quite difficult to implement them. Another potential reason, which may combine with the first, is that algorithms may be described in abstract computational models that may be quite distant from the representation that programmers have of the machines for which they write code.

Keeping with the theme of this article, we illustrate this phenomenon with a relevant case-study: the original presentations of the space-efficient version of the Moebius transform described in [Din21a, §4.3], that we briefly described in Subsection 1.3.

The input is a polynomial of degree d in n variables, represented by a bit array containing all its coefficients. The procedure also depends on a parameter k. The original informal description of the algorithm corresponds to the following pseudo-code.

```
1:  procedure Descent(A, B, C, n, d, k, b)
2:      # A is a bit array containing the coefficient of f (degree d, n variables)
3:      # B and C are large enough scratch arrays
4:      if k = 0 then
5:          Write in C the Algebraic Normal Form of f
6:          Run the classic in-place Moebius transform on C
7:          Do whatever is needed with the truth table of f          ▷ it can be found in C
8:      else
9:          Write in B the coefficients of f with x_n ← b_k          ▷ it has n − 1 variables
10:         Descent(B, A, C, n − 1, k − 1, b)
11:     end if
12: end procedure
13: procedure SpaceEfficientMoebius(A, n, d, k)
14:     # A is a bit array containing the coefficient of f (degree d, n variables)
15:     Allocate a scratch array B of the same size as A
16:     Allocate a scratch array C of size 2^{n−k}
17:     for all b ∈ {0,1}^k do
18:         # Visit the truth table of f with the k last variable specialized to b
19:         Descent(A, B, C, n, d, k, b)
20:     end for
21: end procedure
```

With the right choice of k (that balances the sizes of A, B and C), this requires only about 3 times the space needed to store the input polynomial. The total running time is shown in [Din21a] to be less than $n2^n$ bit operations.

[6] A couple years ago, while working on something very different, the author received an anonymous review that reads: "[the 12-page paper] does not have a big research contribution beyond the straightforward implementations and I don't think it is suitable for publication".

Let us focus on the statement in line 9. We have the coefficients of f in the array `A`, we need to fix the last variable x_n to either zero or one and write the coefficients of the resulting polynomial (let us write it g) in $n-1$ variables into the array `B`. Let m denote a monomial that does not contain x_n. If x_n is fixed to zero, then the coefficient of m in g is the same as the coefficient of m in f. It is therefore sufficient to set `B[j] = A[i]`, where `i` (resp. `j`) denotes the location of the coefficient of the monomial m in the array `A` (resp. `B`).

If, on the other hand, x_n is fixed to one, then the coefficient of m in g is the sum of the coefficients of m and mx_n in f. It is then sufficient to do `B[j] = A[i] ^ A[k]`, where, in addition to the preceding notations, `k` denotes the location of the coefficient of the monomial mx_n in the array `A`. In both case, it may seem that processing each input coefficient requires a simple statement that takes constant time.

The algorithm has originally been described in the computational model of straight-line programs, which originated in computational group theory [BS84], and can be seen as an abstraction of a fixed circuit:

> We estimate the complexity of a straight-line implementation of our algorithm by counting the number of bit operations (e.g., AND, OR, XOR) on pairs of bits. This ignores bookkeeping operations such as moving a bit from one position to another [Din21a, §2.2].

As such, in this computational model, there is simply no need to compute the indices `i`, `j` and `k` at "run-time", because they are in fact "hard-coded" in a straight-line program that implements the Moebius transform for a given input size. The statement `B[j] = A[i] ^ A[k]` costs one bit operation, and the statement `B[j] = A[i]` costs nothing.

However, anyone trying implement this algorithm using usual programming languages would have to deal with the following "implementation detail": *how do we compute the indices* `i`*,* `j` *and* `k`*? What is the minimum overhead that this induces? How should the coefficients be ordered in* `A` *and* `B`*?* This makes several non-trivial algorithmic design decisions to take. It is our belief that these questions are going to leave a decent fraction of the readers scratching their heads for a while.

It is not very straightforward to turn the description of the algorithm into a program for a common model of computation such as the Random Access Machine, let alone the C programming language. And once it will be done, it is not obvious at all that the number of "elementary instructions" executed by this program will match the number of "bit operations" announced in [Din21a].

The heart of the problem seems to be the computational model, which is too abstract and too distant from existing hardware.

The good news is that we provide an in-place Moebius transform algorithm, along with a full C implementation as the litMOB component of the BeanPolE library, with an improved time complexity of $\mathcal{O}(d2^n)$, on real hardware.

2 Using the BeanPolE Library

This section is the official user guide of the BeanPolE library. It is sufficient to include the `BeanPolE.h` header file to access the library functions, in addition to linking against the library itself.

BeanPolE does not dynamically allocate any memory on the heap (it doesn't call `malloc()` or any other equivalent function), and it only uses the stack parsimoniously. This makes it even more obvious that its algorithms work in-place. However, this requires a few global arrays of modest size. Therefore, there are compile-time bounds on the number of variables and the degree of polynomials that can be handled. These bounds are easy to change.

8 ⟨*BeanPolE.h* 8⟩≡

```
#ifndef __BeanPolE_H
#define __BeanPolE_H

#include <stdbool.h>
#include <stddef.h>

#define BeanPolE_MAXN  128                    /* maximum #variables */
#define BeanPolE_MAXD   16                        /* maximum degree */
⟨Global variables declaration 19b⟩
⟨Interface of the library 9a⟩
⟨Interface of the litFES component 10a⟩
⟨Interface of the litMOB component 10b⟩
#endif
```

Invoking any library function with $n > d$, $n >$ `BeanPolE_MAXN` or $d >$ `BeanPolE_MAXD` yields a fatal error.

Before being used, the library must be initialized (this populates its global arrays). Calling any other BeanPolE function before invoking `BeanPolE_init()` is an error and leads to undefined behavior.

9a ⟨Interface of the library 9a⟩≡ (8) 9b▷

```
/*
 * Initialize the library. This takes time O(BeanPolE_MAXN ** 2).
 */
void BeanPolE_init();
```

2.1 Data Structures

The first thing one needs to know to use the library is how Boolean polynomials are represented in memory. BeanPolE uses a simple dense representation with one bit for each possible monomial. A Boolean polynomial of degree at most d in n variables has at most $\binom{n}{\downarrow d} = \sum_{k=0}^{d} \binom{n}{k}$ terms — this convenient notation for the partial sum is borrowed from [Din21a]. Indeed, there are $\binom{n}{k}$ Boolean monomials of degree k: they are the product of k distinct variables. The polynomial f is therefore stored in an array `A` containing $\binom{n}{\downarrow d}$ entries of type `bool`. Note that this requires a C99-capable compiler.

More sophisticated data structures to represent multivariate polynomials in memory are described in the literature. In the Boolean case, the PolyBori [BD09] library uses a solution based on Zero-suppressed binary Decision Diagrams (it is used in SageMath to manipulate Boolean polynomials). Our simpler solution allows constant-time access to the coefficients of a monomial (once the corresponding indices are known), and this turns out to be a crucial advantage.

It is the user's responsibility to allocate the array `A`. To facilitate this, BeanPolE offers a convenience function that returns its size.

9b ⟨Interface of the library 9a⟩+≡ (8) ◁9a

```
/*
 * Return the number of monomials of degree at most d in n variables.
 */
size_t BeanPolE_size(int n, int d);
```

The two components litFES and litMOB use two different monomial orders. Monomial are naturally numbered according to their rank in the relevant order. The i-th entry of the array `A` is the coefficient of the i-th monomial.

litFES uses the *graded lexicographic* order: monomials are ordered by increasing degree, and ties are broken using the lexicographic order (on the exponent vector). This yields

$1 < x_0 < x_1 < x_7 < x_0x_1 < x_4x_5 < x_0x_7 < \ldots$. If f is represented by `A`, then `A[0]` is the constant term of f. The coefficient of x_i is in `A[1 + i]`, etc.

litMOB uses the (plain) lexicographic order. This time we have $1 < x_0 < x_1 < x_0x_1 < x_4x_5 < x_7 < x_0x_7 < \ldots$.

Even though it is not its main feature, the library exposes two functions to evaluate a polynomial on a single arbitrary input. The difference between them lies only on the order in which the coefficients of the polynomial are given.

10a ⟨*Interface of the litFES component* 10a⟩≡ (8) 10c▷

```
/*
 * Return the evaluation of a degree-d polynomial in n variables on
 * input x[].  x[] must be of size at least n.  The coefficients of the
 * polynomial are given in A[], in graded lexicographic order.
 */
bool litFES_eval(int n, int d, const bool A[], const bool x[]);
```

10b ⟨*Interface of the litMOB component* 10b⟩≡ (8) 10d▷

```
/*
 * Return the evaluation of a degree-d polynomial in n variables on
 * input x[].  x[] must be of size at least n.  The coefficients of the
 * polynomial are given in A[], in (plain) lexicographic order.
 */
bool litMOB_eval(int n, int d, const bool A[], const bool x[]);
```

When monomials themselves have to be represented in memory, they are always in "sparse" representation. They are described by their degree and an integer array containing the variables indices (sorted in ascending order). For example, the monomial $x_0x_6x_9$ is represented by the array `[0, 6, 9]`. Note that the numbering of variables starts at zero.

It is the responsibility of the user to populate `A` correctly before invoking the library functions. Convenience ranking functions are provided for this purpose. The demonstration programs use them while parsing the text files containing the polynomials. Notice how n and d play dual roles in these two monomial orders.

10c ⟨*Interface of the litFES component* 10a⟩+≡ (8) ◁10a 11c▷

```
/*
 * Return the rank (in the graded lexicographic order) of the degree-k
 * monomial described by m[], among all monomials in n variables.
 * The array m[] has dimension k and contains the indices of the
 * variables that appear in the monomial, in increasing order.
 * The indices start at zero.  This function runs in time linear in k.
 */
size_t litFES_rank(int n, int k, const int m[]);
```

10d ⟨*Interface of the litMOB component* 10b⟩+≡ (8) ◁10b 12a▷

```
/*
 * Return the rank (in the lexicographic order) of the degree-k
 * monomial described by m[], among all monomials of degree at most d.
 * The array m[] has dimension d and contains the indices of the
 * variables that appear in the monomial, in increasing order. The
 * variable indices start at zero.
 * This function runs in time linear in k.
 */
size_t litMOB_rank(int d, int k, const int m[]);
```

2.2 Iterators

In order to iterate over a sequence of objects (entries of the truth table, monomials, etc.), we use *iterators* made of three functions:

- `xxx_prepare`: initializes the iteration.
- `xxx_advance`: move to the next element.
- `xxx_finished`: indicates whether there is a next element.

These functions can be used as follows:

11a ⟨*Example iteration* 11a⟩≡ 11b▷
```
for (xxx_prepare(); !xxx_finished(); xxx_advance()) {
        ...
}
```

Or equivalently:

11b ⟨*Example iteration* 11a⟩+≡ ◁11a
```
xxx_prepare();
while (!xxx_finished()) {
        ...
        xxx_advance();
}
```

Because it could be useful in some applications, the library exposes two iterators over monomials of degree exactly k (resp. at most k). Both iterators maintain a state that consists of a single integer. The initial value of this extra variable is provided by the preparation function.

11c ⟨*Interface of the litFES component* 10a⟩+≡ (8) ◁10c 11d▷
```
/*
 * Prepare the enumeration of all degree-d monomials in n variables in
 * lexicographic order.  m[] must be of size at least d + 2.
 * The indices of the variable in the first monomial can be read in m[].
 * This function runs in time O(d).
 */
int litFES_combination_prepare(int n, int d, int m[]);
```

11d ⟨*Interface of the litFES component* 10a⟩+≡ (8) ◁11c 11e▷
```
/*
 * Advance to the next degree-d monomial in n variables in lexicographic
 * order.  The next monomial can be read in m[] and the return value
 * must be used as the j argument in the next invocation.
 * This functions runs in constant amortized time.
 */
int litFES_combination_advance(int n, int d, int m[], int j);
```

11e ⟨*Interface of the litFES component* 10a⟩+≡ (8) ◁11d 12d▷
```
/*
 * Return 1 if there is no next monomial.
 */
bool litFES_combination_finished(int n, int j);
```

In order to enumerate monomials of degree at most d in n variables, we must know the degree of the current monomial. The corresponding iterator maintains this degree in an external variable `k`:

12a ⟨*Interface of the litMOB component* 10b⟩+≡ (8) ◁10d 12b▷

```
/*
 * Prepare the enumeration of all monomials of degree at most d in
 * lexicographic order.  m[] must be of size at least d + 1.
 * The return value of this function is the degree of the first monomial
 * (zero).  This function runs in constant time.
 */
int litMOB_combination_prepare(int d, int m[]);
```

12b ⟨*Interface of the litMOB component* 10b⟩+≡ (8) ◁12a 12c▷

```
/*
 * Advance m[] to the next monomial of degree at most d in lexicographic
 * order.  This function runs in constant amortized time.  k is the
 * degree of the current monomial.  The return value is the degree of
 * the next monomial (the new value of k).  The next monomial can be
 * found in m[d-k:d].
 */
int litMOB_combination_advance(int d, int k, int m[]);
```

12c ⟨*Interface of the litMOB component* 10b⟩+≡ (8) ◁12b 14a▷

```
/*
 * Return 1 if there is no next monomial.
 */
bool litMOB_combination_finished(int n, int d, int m[]);
```

2.3 The litFES Component

This is an implementation of the FES algorithm. This algorithm visits the entries of the truth table one at a time, sequentially. It jumps from an entry to the next in $\mathcal{O}(d)$ operations in the worst case.

The litFES component exposes a few functions that have a common set of arguments. They are now described once and for all.

12d ⟨*Interface of the litFES component* 10a⟩+≡ (8) ◁11e 12e▷

```
/*
 * Functions of the litFES component take the following arguments:
 * n  : number of variables of the polynomial
 * d  : degree of the polynomial
 * A[]: coefficients of the polynomial in graded lexicographic order
 * x[]: variables on which f is evaluated. x[] has dimension n
 * s  : internal state of the enumeration procedure (opaque)
 */
```

This provides the semantics of the algorithm.

12e ⟨*Interface of the litFES component* 10a⟩+≡ (8) ◁12d 13a▷

```
/*
 * At all times, litFES maintains the following invariant:
 * A[0] contains the result of the evaluation of the polynomial on x
 */
```

The litFES functions maintain an internal state, which is an (opaque) object containing $1 + 2 \times$ `LITFES_MAXN` integers.

13a ⟨*Interface of the litFES component* 10a⟩+≡ (8) ◁12e 13b▷
```
struct litFES_state {
        int h;                                  /* height of the stack */
        int stack[BeanPolE_MAXN + 1];
        int f[BeanPolE_MAXN + 1];                    /* focus pointers */
};
```

The four main functions that implement the FES algorithm are the following:

13b ⟨*Interface of the litFES component* 10a⟩+≡ (8) ◁13a 13c▷
```
/*
 * Run the required preparations. Modify A[] and sets x[] to
 * (0, 0, ..., 0). x[] must have size n + 1.
 * The running time of this functions is at most d**2 * phi**d times the
 * size of A, where phi == (1 + sqrt(5)) / 2 is the golden ratio.
 */
void litFES_prepare(int n, int d, bool A[], bool x[],
                    struct litFES_state *s);
```

13c ⟨*Interface of the litFES component* 10a⟩+≡ (8) ◁13b 13d▷
```
/*
 * Advance x[] to the next n-bit string and update A[0] accordingly.
 * This functions executes O(d) elementary instructions.
 * It modifies A[].  litFES_prepare() must have been called beforehand.
 */
void litFES_advance(int n, int d, bool A[], bool x[],
                    struct litFES_state *s);
```

13d ⟨*Interface of the litFES component* 10a⟩+≡ (8) ◁13c 13e▷
```
/*
 * Return 1 if all entries of the truth table have been visited.
 * This happens after 2**n calls to litFES_advance().
 * This function runs in constant time.
 */
bool litFES_finished(int n, const bool x[]);
```

13e ⟨*Interface of the litFES component* 10a⟩+≡ (8) ◁13d
```
/*
 * Restore A[] to its initial value -- before the call to
 * litFES_prepare().
 * This assumes that litFES_finished() has returned 1.
 * Invoking this function is optional.  Its running time is about the
 * same as that of libFES_prepare().
 */
void litFES_restore(int n, int d, bool A[], struct litFES_state *s);
```

It must be noted that litFES is thread-safe: if two threads "own" two different polynomials and two different `litFES_state` objects, they can invoke the library functions concurrently on their own arguments.

2.4 The litMOB Component

This is an implementation of the in-place Moebius transform. litMOB produces the entries of the truth table in *chunks* of size 2^d (where d is the degree of the polynomial). The algorithm visits the 2^{n-d} chunks of the truth table one at a time, sequentially. It jumps from a chunk to the next with an amortized cost of $\mathcal{O}\left(d2^d\right)$ operations.

The semantics of the functions are given by:

14a ⟨*Interface of the litMOB component* 10b⟩+≡ (8) ◁12c 14b▷

```
/*
 * litMOB maintains the following invariant:
 * The first 2**d entries of A contains the truth table of f where
 * the last n-d variables are fixed to x.
 */
```

14b ⟨*Interface of the litMOB component* 10b⟩+≡ (8) ◁14a 14c▷

```
/*
 * Run the required preparations. Sets x[] to (0, 0, ..., 0).
 * x[] must have size at least n + 1.
 * The first chunk of the truth table is available in A[0:2**d] on exit.
 * This functions runs in time O(d * 2**d).
 */
void litMOB_prepare(int n, int d, bool A[], bool x[]);
```

14c ⟨*Interface of the litMOB component* 10b⟩+≡ (8) ◁14b 14d▷

```
/*
 * Advance x[d:n] to the next value in lexicographic order, and update
 * A[0:2**d] accordingly.  The rest of A[] may be modified.  x[0:d] is
 * not accessed.
 * This functions runs in amortized time O(d * 2**d).
 */
void litMOB_advance(int n, int d, bool A[], bool x[]);
```

14d ⟨*Interface of the litMOB component* 10b⟩+≡ (8) ◁14c

```
/*
 * Return 1 when the whole truth table has been visited.
 * In this case, A[] has been returned to its original state.
 * This happens after 2**(n - d) calls to litMOB() advance().
 * This function runs in constant time.
 */
bool litMOB_finished(int n, const bool x[]);
```

2.5 Demonstration Programs

The companion demonstration programs load a Boolean polynomial f from a text file and print out its full truth table. They are only partially described in this article; how the polynomial is parsed is not relevant, and it actually requires much more code than the actual algorithm to iterate through its truth table. We used these demonstration programs to test the correctness of the library while preparing this work.

14e ⟨*Main loop of the litFES demonstration program* 14e⟩≡

```
bool x[n + 1];
struct litFES_state state;
litFES_prepare(n, d, A, x, &state);
while (!litFES_finished(n, x)) {
```

```
        for (int i = 0; i < n; i++)                          /* print x */
                printf("%01d", x[i]);
        printf(": %d\n", A[0]);                            /* print f(x) */
        litFES_advance(n, d, A, x, &state);
}
litFES_restore(n, d, A, x, &state);
```

The loop visits each entry of the truth table. For each $x \in \{0,1\}^n$, it prints x and $f(x)$, which is available in `A[0]` according to the invariant maintained by the library. As per the specification of `litFES_advance`, the loop exits during the 2^n-th iteration. The total time spent in `litFES_advance` is $\mathcal{O}(d2^n)$. The `A` array is returned into its original state.

The corresponding program for litMOB prints a chunk of size 2^d of the truth table in each iteration.

15 ⟨*Main loop of the litMOB demonstration program* 15⟩≡

```
bool x[n + 1];
litMOB_prepare(n, d, A, x);
while (!litMOB_finished(n, x)) {
        int k = 0;                  /* print current truth table chunk */
        for (;;) {
                for (int i = 0; i < n; i++)                  /* print x */
                        printf("%01d", x[i]);
                printf(": %d\n", A[k]);                    /* print f(x) */
                int j = 0;                                  /* advance x */
                while (j < d && x[j] == 1) {
                        x[j] = 0;
                        j += 1;
                }
                if (j == d)
                        break;
                x[j] = 1;
                k += 1;
        }
        litMOB_advance(n, d, A, x);
}
```

This concludes the user guide of the BeanPolE library. The rest of this paper describes its implementation.

3 Preliminaries

3.1 Bit strings and Integers

Bit strings are elements of $\{0,1\}^*$. The i-th symbol of a bit string x is denoted as x_i. We simply denote by xy the concatenation of two bit strings x and y. Similarly, if a is a bit string, we write $a^k = aaa\ldots a$ where a is repeated k times. We write $x \subseteq y$ and we say that "x is contained in y" when $x_i = 1 \Rightarrow y_i = 1$ (this is equivalent to $x\&y = x$).

Any non-negative integer can be written in base two as $i = (\ldots a_3a_2a_1a_0)_2 = \sum_k a_k 2^k$, where a is a bit string. The right shift operator is defined by $i \gg 1 = \lfloor i/2 \rfloor$. Clearly, if $i = (\ldots a_2a_1a_0)_2$, then $i \gg 1 = (\ldots a_2a_1)_2$.

The bitwise AND and XOR operators that act naturally on bit strings are naturally extended to the integers (applying them to their base-2 representation). They are denoted by $a\&b$ and $a \oplus b$. The bitwise NOT operator is denoted as $\overline{x}$.

Let $\rho(i)$ denote the greatest integer k such that 2^k divides i, with $\rho(0) = +\infty$. This locates the rightmost "1" bit in the binary representation of the integer i.

Define the function $\rho^* : (\dots a_2 a_1 a_0)2 \mapsto \{j \in \mathbb{N} \ : \ a_j = 1\}$. This tracks the location of all "1" bits in the binary representation of i. It follows that $\rho(i) = \min \rho^*(i)$. For instance, $42 = (101010)_2$, therefore $\rho^*(42) = \{1, 3, 5\}$. Also, $1337 = (10100111001)_2$, and $\rho^*(1337) = \{0, 3, 4, 5, 8, 10\}$.

3.2 Boolean Monomials and Boolean Polynomials

The ring of (Boolean) polynomials in n variables $x = (x_0, \dots, x_{n-1})$, hereafter denoted by $\mathbb{B}$, is the quotient of the polynomial ring $\mathbb{F}_2[x_0, \dots, x_{n-1}]$ by the ideal spanned by the "field equations" $\langle x_0^2 - x_0, \dots, x_{n-1}^2 - x_{n-1} \rangle$. Therefore, if f is a Boolean polynomial, then the exponent of any variable in all monomials is either 0 or 1.

A Boolean monomial ${x_0}^{e_0} {x_1}^{e_1} \dots {x_{n-1}}^{e_{n-1}}$ is completely described by the bit string $e_0 \dots e_{n-1}$ (the exponent vector). It is also completely determined by the set $\{0 \leq i < n \ : \ e_i = 1\}$. Therefore, we happily identify monomials with n-bit strings with subsets of $\{0, \dots, n-1\}$.

The degree of a monomial is the Hamming weight of the exponent bit string (or the cardinality of the set of exponents). The degree of a Boolean polynomial is the largest degree of its monomials.

3.3 Binomial Coefficients

Let H denote the binary entropy function, meaning that

$$H(x) = -x \log_2(x) - (1-x) \log_2(1-x),$$

for all $0 < x < 1$. It can be continuously extended to $H(0) = H(1) = 0$. The following standard bounds for the binomial coefficient can be derived from Stirling's formula:

$$\frac{2^{nH(x)}}{\sqrt{8nx(1-x)}} \leq \binom{n}{xn} \leq \frac{2^{nH(x)}}{\sqrt{2\pi nx(1-x)}}, \qquad (0 < x < 1/2) \qquad (1)$$

$$\binom{n}{\downarrow xn} \leq 2^{nH(x)}, \qquad (0 < x < 1/2) \qquad (2)$$

In addition, when $k \leq n/2$, we have the trivial upper-bound:

$$\binom{n}{k} \leq \binom{n}{\downarrow k} \leq k \binom{n}{k}, \qquad (0 < x < 1/2) \qquad (3)$$

We will later need the following technical result.

Lemma 1. *For any $d \in \mathbb{N}$, we have*

$$\sum_{i=0}^{+\infty} 2^{-i} \binom{i}{d} = 2.$$

Proof. To begin with, it is clear that the series converges, if only because $\binom{i}{d} = \mathcal{O}\left(i^d\right)$.

We establish the result using the method of "creative telescoping". Define $F(d, i) = 2^{-i}\binom{i}{d}$ and $G(d, i) = 2\frac{d-i}{d+1} F(i, d)$. Then we have

$$F(d+1, i) - F(d, i) = G(d, i+1) - G(d, i).$$

Summing on $0 \leq i \leq n$ yields

$$\sum_{i=0}^{n} F(d+1,i) - \sum_{i=0}^{n} F(d,i) = G(d,n+1) - G(d,0).$$

In all cases, $G(d,0) = 0$. Then passing to the limit with $n \to +\infty$, we find that:

$$\sum_{i=0}^{+\infty} F(d+1,i) - \sum_{i=0}^{+\infty} F(d,i) = \lim_{n\to+\infty} G(d,n+1) = 0.$$

This shows that the sum of the series is independent of d. Then with $d = 0$ we quickly find that

$$\sum_{i=0}^{+\infty} F(0,i) = \sum_{i=0}^{+\infty} 2^{-i} = 2$$

□

3.4 Derivatives

By analogy with the corresponding notion from calculus, define the "differential operator" $D_k : \mathbb{B} \mapsto \mathbb{B}$ that differentiates with respect to the k-th variable as

$$D_k : f(x) \mapsto f(x_0, \ldots, x_{k-1}, x_k + 1, x_{k+1}, \ldots, x_{n-1}) + f(x)$$

This is clearly a linear operator on $\mathbb{B}$. If m denotes a monomial that is not a multiple of x_k, then it is easy to check that $D_k(m) = 0$ and $D_k(mx_k) = m$. These rules make it straightforward to evaluate the derivatives of any polynomial. In fact, this shows that $D_k(f)$ contains all the monomials of f that are divisible by x_k, divided by x_k. It follows that $D_k(f)$ does not depend on x_k and that $\deg D_k(f) = \max(0, \deg f - 1)$. More precisely, we can write

$$f(x) = f\left(x_0, \ldots, x_{k-1}, 0, x_{k+1}, \ldots, x_{n-1}\right) + x_k \cdot D_k(f)(x)$$

Higher-order derivatives are simply the derivatives of the derivatives. For instance

$$D_\ell \circ D_k : f \mapsto (f(x + x_k + x_\ell) + f(x + x_\ell)) + (f(x + x_k) + f(x))$$

Looking at this definition, it is easy to conclude that the differentiation operators commute: $D_k \circ D_\ell = D_\ell \circ D_k$. This implies that differentiating with respect to a subset of the variables (*i.e.* a monomial) makes sense. We thus write $D_m(f)$ for an arbitrary monomial m. If f has degree d, then its d-th derivatives are constant. The Boolean polynomial $D_m(f)$ contains all monomials of f that are multiples of m, divided by m. It follows that evaluating $D_m(f)$ on zero yields the coefficient of the monomial m in f. And because $D_m(f)$ does not depend on the variables contained in m, then the same result is obtained by evaluating $D_m(f)$ on m.

3.5 Organization of the Library

The implementation of the library consists of a a few C files, where "private" internal functions are clearly separated from "public" functions that are exposed in the interface. Functions common to multiple components go in `BeanPole.c`.

17 ⟨*BeanPolE.c* 17⟩≡

```
#include <assert.h>
#include "BeanPolE.h"
⟨Global variables 19a⟩
⟨Public functions 20a⟩
```

4 Tracking Bits in a Counter

In this section, we describe a *loopless* (*i.e.* worst-case constant-time) algorithm to evaluate ρ^* on consecutive integers, starting from zero. In other terms, we track the positions of all set bits in a counter i as it is incremented.

This can be done in a loopless way using a stack. This stack contains, from bottom to top, the locations of all non-zero bits of the counter, from the leftmost (at the bottom of the stack) to rightmost (at the top). If the counter is zero, the stack is empty (denoted by `[]`). If the counter is, say, $(101010)_2$, then the stack is `[5, 3, 1]`. If the counter is incremented to $(101011)_2$, the stack becomes `[5, 3, 1, 0]`.

Here is how to update the stack when the counter is incremented. The counter can always be written $i = (a01^k)_2$ for some $k \geq 0$ and some bit string a. The next value is $i + 1 = (a10^k)_2$. Note that $k = \rho(i + 1)$ is in fact the position of the rightmost non-zero bit of $i + 1$. To maintain the stack while i is incremented, we need to 1) pop the top k entries and 2) push k.

The problem boils down to evaluate ρ on consecutive integers, starting from zero. Some processors have a builtin instruction to evaluate ρ. For instance, the x86-compatible CPUs have the `bsf` instruction that does just this. The Gnu C Compiler exposes the `__builtin_ffs()` pseudo-function that invokes the corresponding instruction if the target architecture has it. On the other hand, trying to evaluating ρ efficiently without relying on *ad hoc* hardware instructions leads to a wealth of "bitwise tricks", "hacker's delight" and whatnot.

A particularly elegant solution has been proposed by Ehrlich in 1973 [Ehr73, BER76] (see also [Knu14, §7.2.1.1]). Using an array of $(n + 1)$ "focus pointers", it enables the evaluation of ρ on the next value of the counter in a constant number of operations.

The opaque `litFES_state` type described in the library interface in fact contains the persistent state needed by this procedure. The value of the counter is not stored, because it is represented implicitly by the stack and the focus pointers.

18a ⟨*Prepare the* `litFES_state` 18a⟩≡ (27a)

```
s->h = 0;
for (int i = 0; i < n + 1; i++)
        s->f[i] = i;
```

The loopless procedure to maintain the state when i is incremented is:

18b ⟨*Increment* i*; maintain* $\rho(i)$ *in* `k`*; maintain* $\rho^*(i)$ *in* `stack[0:h]` 18b⟩≡ (27b)

```
int k = f[0];                                   /* k == rho(i) */
f[0] = 0;                                /* update focus pointers */
f[k] = f[k + 1];
f[k + 1] = k + 1;
h -= k;                                     /* pop top k elements */
stack[h] = k;                            /* push k onto the stack */
h += 1;
```

Note that, as a by-product, what is described in this section can be seen as an algorithm that increments an n-bit counter in *worst-case* constant time. In contrast, the classic solution of propagating carries works in *amortized* constant time only. If w-bit registers are available, then this works for counters values of up to 2^w bits, using $2 + 2^{w+1}$ words of storage. Of course, the binary representation of the counter is not maintained, but an alternate (redundant) representation is.

5 Operations on Monomials

5.1 Ranking

It is not very difficult to compute the rank of a given degree-k monomial among all monomials of degree at most d in n variables, in the orders that matter to us. All this is well-known (see for instance [Rus03]). The rank of m is the number of strictly smaller monomials. A ranking function assigns a distinct positive integer to each monomial.

Let $\mu_{\leq d}$ denote the ranking function for monomials of degree at most d in the plain lexicographic order. Monomials are compared by looking at their greatest variable first: mx_n is greater than all monomials that only contain $x_1, \ldots, x_{n-1}$.

Let us begin with the plain lexicographic order restricted to degree-k monomials. Let μ_k denote the corresponding ranking function, which assigns an integer less than $\binom{n}{k}$ to each monomial of degree k in n variables. If $m = \{i_0, \ldots, i_{k-1}\}$, then m is greater than all degree-k monomials only containing variables strictly smaller than i_{k-1}. There are $\binom{i_{k-1}}{k}$ such monomials (recall that variable numbering starts at zero). Then, the rank of m among all degree-k monomials whose greatest variable is i_{k-1} is precisely $\mu_{k-1}(i_0, \ldots, i_{k-2})$. This leads to the recursive definition:

$$\mu_0(\emptyset) := 0$$
$$\mu_k(\{i_0, \ldots, i_{k-1}\}) := \binom{i_{k-1}}{k} + \mu_{k-1}(i_0, \ldots, i_{k-2})$$

Let $\mu_{\downarrow d}$ denote the the ranking function for all monomials of degree at most d in n variables in the plain lexicographic order. It assigns integer less than $\binom{n}{\downarrow d}$ to each of them. A similar reasoning shows that:

$$\mu_{\downarrow 0}(\emptyset) := 0$$
$$\mu_{\downarrow d}(\{i_0, \ldots, i_{k-1}\}) := \binom{i_k}{\downarrow d} + \mu_{\downarrow d-1}(i_0, \ldots, i_{k-2})$$

Finally, let ν denote the ranking function for the graded lexicographic order. Because monomial are ordered by degree, then $\nu(m) < \nu(m')$ if $\deg m < \deg m'$. If a monomial m has degree k, there are $\binom{n}{\downarrow k-1}$ monomials of strictly smaller degree. This shows that:

$$\nu(m) := \texttt{let } k = \deg m \texttt{ in } \binom{n}{\downarrow k-1} + \mu_k(m).$$

Evaluating μ_k (and thus ν) or $\mu_\downarrow$ requires k operations, assuming that the binomials coefficients and their partial sums are precomputed. To make this work, we define two global bi-dimensional arrays: `BeanPolE_N[n][k]` contains $\binom{n}{\downarrow k-1}$ (the number of monomials of degree strictly less than k in n variables) and `BeanPolE_binomial[n][k]` simply contains the binomial coefficient $\binom{n}{k}$.

19a ⟨*Global variables* 19a⟩≡ (17)
```
size_t BeanPolE_binomial[BeanPolE_MAXN + 1][BeanPolE_MAXD + 2];
size_t BeanPolE_N[BeanPolE_MAXN + 1][BeanPolE_MAXD + 2];
```

19b ⟨*Global variables declaration* 19b⟩≡ (8)
```
extern size_t BeanPolE_binomial[BeanPolE_MAXN + 1][BeanPolE_MAXD + 2];
extern size_t BeanPolE_N[BeanPolE_MAXN + 1][BeanPolE_MAXD + 2];
```

Unfolding both recursive definitions leads to the sums:

$$\mu_d(\{i_0,\ldots,i_{k-1}\}) = \sum_{j=0}^{k-1} \binom{i_j}{j+1}$$
$$\mu_{\downarrow d}(\{i_0,\ldots,i_{k-1}\}) = \sum_{j=0}^{k-1} \binom{i_j}{\downarrow d-k+j+1}$$

This being said, writing the code of the convenience ranking functions is now completely straightforward.

20a ⟨*Public functions* 20a⟩≡ (17) 21a▷
```
size_t BeanPolE_size(int n, int d)
{
        return BeanPolE_N[n][d + 1];
}
```

20b ⟨*litFES Public functions* 20b⟩≡ (25a) 21b▷
```
size_t litFES_rank(int n, int k, const int m[])
{
        size_t rank = BeanPolE_N[n][k];
        for (int j = 0; j < k; j++)
                rank += BeanPolE_binomial[m[j]][j + 1];
        return rank;
}
```

20c ⟨*litMOB Public functions* 20c⟩≡ (31) 22b▷
```
size_t litMOB_rank(int d, int k, const int m[])
{
        size_t rank = 0;
        for (int j = 0; j < k; j++)
                rank += BeanPolE_N[m[j]][d + 2 - k + j];
        return rank;
}
```

Note that computing $\mu_k(\{i_0,\ldots,i_{k-1}\})$ yields in passing the values of

$$\mu_1(\{i_0\}), \mu_2(\{i_0,i_1\}), \ldots, \mu_k(\{i_0,\ldots,i_{k-1}\}).$$

This naturally extends to ν. In other terms, it is possible to compute the rank (in the graded lexicographic order) of each *prefix* of the original monomial in $\mathcal{O}(k)$ operations. This detail plays a crucial role in the implementation of litFES in section 6.2.

In a dual way, suppose that $r = \mu_{\downarrow d'}(\{i_0,\ldots,i_{k-1}\})$. It is easy to update r if the smallest variable is removed, or if another even smaller variable $j < i_0$ is added:

$$\mu_{\downarrow d}(\{i_1,\ldots,i_{k-1}\}) = r - \binom{i_0}{\downarrow d-k+1} \tag{4}$$
$$\mu_{\downarrow d}(\{j,i_0,\ldots,i_{k-1}\}) = r + \binom{j}{\downarrow d-k} \tag{5}$$

The ability to update / downdate the rank of a monomial in constant time plays a crucial role in the implementation of litMOB in section 7.

Lastly, the binomial coefficients need to be computed before being used. This is the purpose of the initialization function.

21a ⟨*Public functions* 20a⟩+≡ (17) ◁20a

```
void BeanPolE_init()
{
        size_t pascal[BeanPolE_MAXN + 1];
        for (int n = 0; n <= BeanPolE_MAXN; n++)
                pascal[n] = 0;
        pascal[0] = 1;
        for (int n = 0; n <= BeanPolE_MAXN; n++) {
                pascal[n] = 1;
                for (int k = n-1; k > 0; k-)
                        pascal[k] += pascal[k - 1];
                for (int k = 0; k <= BeanPolE_MAXD; k++)
                        BeanPolE_binomial[n][k] = pascal[k];
                size_t acc = 0;
                for (int k = 0; k <= BeanPolE_MAXD + 1; k++) {
                        BeanPolE_N[n][k] = acc;
                        acc += BeanPolE_binomial[n][k];
                }
        }
}
```

5.2 Iteration

This section describes two iterators over the set of monomials in lexicographic order. One enumerates degree-d monomials, while the other lists monomials of degree at most d.

5.2.1 Degree-k Monomials

Enumerate all degree-k monomials in lexicographic order is strictly equivalent to the lexicographic enumeration of all k-subsets of $\{0, 1, 2, \ldots, n-1\}$. For this, we use the venerable algorithm first described by Mifsud [Mif63], along with the optimization described by Dvořák in [Dvo90]. All of this is summarized in algorithm T from [Knu14, §7.2.1.3].

The iterator maintains an integer variable j across invocations. The initial value of j is the return value of `litFES_combination_prepare`. j is the smallest index such that `m[j + 1] > j + 1`.

21b ⟨*litFES Public functions* 20b⟩+≡ (25a) ◁20b ?? ▷

```
int litFES_combination_prepare(int n, int k, int x[])
{
        for (int i = 0; i < k; i++)
                x[i] = i;
        x[k] = n;
        x[k + 1] = 0;
        return k - 1;
}
```

21c ⟨*litFES Internal functions* 21c⟩≡ (25a) 22a ▷

```
int litFES_combination_advance(int n, int k, int m[], int j)
{
        if (n == k)                        /* Easy cases with O(1) cost */
                return n;
```

```
        if (j >= 0) {
                m[j] = j + 1;
                return j - 1;
        }
        if (m[0] + 1 < m[1]) {
                m[0] += 1;
                return j;
        }
        j = 0;                          /* Hard case. Must locate j again */
        while (m[j] + 1 == m[j + 1]) {
                m[j] = j;
                j += 1;
        }
        if (j >= k)
                return n;
        m[j] = m[j] + 1;
        return j - 1;
}
```

22a ⟨*litFES Internal functions* 21c⟩+≡ (25a) ◁21c 24▷

```
bool litFES_combination_finished(int n, int j)
{
        return j >= n;
}
```

We first note that if the `j = 0` statement is executed, then there will be at least one iteration of the while loop (otherwise the function would have exited in the third `if` statement).

It is not difficult to check that `litFES_combination_advance` runs in constant amortized time, for instance using the accounting method. Each time the function is invoked, two credits must be deposited. Executing the first two `if` statements cost one credit. Executing the rest of the function costs one more credit, plus one extra credit per iteration of the `while` loop after the first one.

We claim that the sum of j and the balance of the account is always equal to $k - 1$. When `litFES_combination_advance` returns, this claim obviously holds. If $j \geq 0$, then running `litFES_combination_advance` increases the account's balance by one credit and j is decremented. If the function terminates in the third `if` statement, both the balance of the account and j are left unchanged. If there are t iterations of the `while` loop, then $t - 1$ credits are withdrawn from the account and j is increased by $t - 1$.

Because $-1 \leq j \leq k$, it follows that the balance of the account is lower-bounded by -1. This establishes the constant amortized complexity.

5.2.2 Monomials of Degree at Most k

The next couple of functions constitute the main technical ingredient of litMOB. They enumerate all monomials m of degree at most d in lexicographic order, which is fairly classic (we could not trace a precise reference).

22b ⟨*litMOB Public functions* 20c⟩+≡ (31) ◁20c 23c▷

```
int litMOB_combination_prepare(int d, int m[])
{
        if (d > 0)
                m[d - 1] = 0;
        m[d] = 1;
```

```
        return 0;
}
```

23a ⟨*litMOB Internal functions* 23a⟩≡ (31) 23b▷

```
int litMOB_combination_advance(int d, int k, int m[])
{
        /* Can we add a zero to the left? */
        if (k != d && m[d - k] != 0) {
                k += 1;
                m[d - k] = 0;
                return k;
        }
        /* Erase leftmost digits that cannot be incremented */
        while (k > 1 && m[d-k] + 1 == m[d-k+1])
                k -= 1;
        /* Increment leftmost digit */
        m[d - k] += 1;
        return k;
}
```

It is easy to show that this function runs in constant amortized time. By the accounting method, assume that each call to `litMOB_combination_advance` requires a deposit of 2 credits. The first `if` statement costs one credit. Each iteration of the `while` loop costs one credit. The final chunk of the function costs one additional credit. If a zero is added (the condition in first `if` statement is true), then the function returns quickly and one extra credit is left in the account. Otherwise, t iterations of the `while` loop take place, t items are erased from the current tuple and t credits are withdrawn from the account. Thus, the balance of the account is exactly the degree of the current monomial, and as such it cannot become negative.

23b ⟨*litMOB Internal functions* 23a⟩+≡ (31) ◁23a 32a▷

```
bool litMOB_combination_finished(int n, int d, int m[])
{
        return ((d == 0) && (m[d] > 1)) || ((d > 0) && (m[d-1] >= n));
}
```

5.2.3 Application: Single-Point Evaluation

As a first direct application of these monomial enumeration techniques, we provide the procedures that evaluates a polynomial on a single arbitrary input. The value of $f(x)$ is the sum of the coefficients of all monomials that evaluate to 1 on x. These monomials are precisely those that are *contained* in x.

23c ⟨*litMOB Public functions* 20c⟩+≡ (31) ◁22b 33b▷

```
bool litMOB_eval(int n, int d, const bool A[], const bool x[])
{
        /* convert input to sparse representation */
        int xx[n];
        int l = 0;                                          /* k == |x| */
        for (int i = 0; i < n; i++)
                if (x[i])
                        xx[l++] = i;
        bool y = 0;
        int hi = d;
        if (hi > l)
```

```
                hi = l;
        /* Enumerate all subsets of x[] of size at most hi */
        int m[hi + 1];
        int k = litMOB_combination_prepare(hi, m);        /* l == |m| */
        while (!litMOB_combination_finished(l, hi, m)) {
                /* expand the subset */
                int mm[l];
                for (int j = 0; j < k; j++)
                        mm[j] = xx[m[hi - k + j]];
                /* add the coefficient of mm to the result */
                size_t j = litMOB_rank(d, k, mm);
                y ^= A[j];
                k = litMOB_combination_advance(hi, k, m);
        }
        return y;
}
```

A single iteration of the look requires $\mathcal{O}(\ell)$ operations, and $\ell \leq \texttt{hi} = \min(d, k)$. The total number of operations is then $k2^k$ when $k \leq d$ (the input x has low Hamming weight) or simply d times the size of the polynomial when $k \geq d$ (the input x is not sufficiently sparse).

The above function assumes that the input polynomial is in plain lexicographic order. If, on the contrary, the input polynomial is in graded lexicographic order, then it is sufficient to replace `litMOB_rank` by `litFES_rank` in the above function.

5.3 Multiplication

Multiplying two monomials comes down to merging the two sorted arrays that represent them. A function similar to this one can be found in most implementations of the merge sort algorithm; this one is particularly simple because the two arrays contain disjoint sets of values.

24 ⟨*litFES Internal functions* 21c⟩+≡ (25a) ◁22a 25b▷

```
/*
 * Compute the product of two **relatively prime** monomials.
 * Input are in (alen, ax) and (blen, bx).  The return value is the
 * degree of the product.  The output is written in cx (which must be
 * preallocated).  This function runs in time O(alen + blen).
 */
static int merge(int alen, const int ax[], int blen, const int bx[],
                 int cx[])
{
        int i = 0, j = 0;
        for (int k = 0; k < alen + blen; k++) {
                if (j == blen || ax[i] < bx[j])
                        cx[k] = ax[i++];
                else
                        cx[k] = bx[j++];
        }
        return alen + blen;
}
```

6 Implementation of the litFES component

This section implemented an unrolled (non-recursive) version of the Fast Exhaustive Search algorithm, for any degree. It builds upon the recursive presentation given in [BCC+10a], and more precisely in its extended version [BCC+10b]. The algorithm is given in an unrolled version in [Bou11]. Here, its implementation is contained in `litFES.c`:

25a ⟨*litFES.c* 25a⟩≡
```
#include <assert.h>
#include "BeanPolE.h"
⟨litFES Internal functions 21c⟩
⟨litFES Public functions 20b⟩
```

The fact that this algorithm is correct, namely, that it actually compute the truth table, is not obvious. The fact that the version given here is correct directly follows from the correctness proof of the original algorithm given in [Bou11], that we will not repeat.

6.1 Setup Phase

Before the main loop that walks through the truth table, some preparations are in order. When the algorithm starts, the array `A` gives the coefficient of each monomial. By a complete abuse of notation, we allow ourselves to write `A[m]` when m is a monomial to denote `A[i]` where i is the rank of m. This being said, it follows from the discussion at the end of Subsection 3.4 that for each monomial m of degree less than D, `A[m]` contains the result of the evaluation of $D_m(f)$ on m.

The setup phase consists in modifying `A` such that `A[m]` contains the result of the evaluation of $D_m(f)$ on $m \oplus (m \gg 1)$. This is necessary because the bit strings x over which f is evaluated are enumerated in Gray code order; the i-th bit string is $i \oplus (i \gg 1)$.

Because $D_m(f)$ does not depend on the variables in m, we can restrict ourselves to evaluate each $D_m(f)$ on $(m \gg 1)\&\overline{m}$. Note that this monomial has degree less than or equal to that of m. The following function compute this new monomial from m.

25b ⟨*litFES Internal functions* 21c⟩+≡ (25a) ◁24 26a▷
```
/*
 * Given a monomial m of degree k, returns (m » 1) & ~m.
 * Write it in mshift and return its degree.
 */
static int shift_monomial(int k, const int m[], int mshift[])
{
        int j = 0;
        for (int i = 0; i < k; i++) {
                if ((m[i] == 0) || ((i > 0) && (m[i] - 1 == m[i - 1])))
                        continue;
                mshift[j] = m[i] - 1;
                j += 1;
        }
        return j;
}
```

To evaluate $D_m(f)$ on an arbitrary monomial m', we enumerate the monomials of $D_m(f)$ that evaluate to 1 on input m'. Such a monomial:

i) Is a multiple of m — appears in $D_m(f)$.

ii) Divides mm' — $D_m(f)$ evaluate to 1 on m'.

We therefore enumerate all subsets of $m' \& \overline{m}$ of degree less than $d - \deg m$ and multiply them by m (as usual d is the degree of the polynomial f).

26a ⟨*litFES Internal functions* 21c⟩+≡ (25a) ◁25b

```
/*
 * Evaluate D_m(f) on m ^ (m » 1) and store the result in A[i].
 * The monomial m has degree k.  The running time of this function is
 * less than k times the number of monomials of degree at most min(d-k, k)
 * in k variables.
 */
static void tweak_derivative(int n, int d, bool A[], size_t i,
                             int k, const int m[])
{
        /* compute the shifted monomial */
        int mprime[n];
        int l = shift_monomial(k, m, mprime);
        int hi = d - k;                 /* maximum degree of subsets */
        if (l < hi)
                hi = l;
        for (int t = 1; t <= hi; t++) {
                /* Enumerate all degree-t subsets of m' */
                int c[l + 2];
                int it = litFES_combination_prepare(l, t, c);
                while (!litFES_combination_finished(l, it)) {
                        int uu[n];
                        int cc[n];
                        /* compute the product m*m' */
                        for (int j = 0; j < t; j++)
                                cc[j] = mprime[c[j]];
                        merge(k, m, t, cc, uu);
                        /* add the coefficient of m*m' to the result */
                        size_t j = litFES_rank(n, k + t, uu);
                        A[i] ^= A[j];
                        it = litFES_combination_advance(l, t, c, it);
                }
        }
}
```

It is not difficult to see that `tweak_derivative` is involutive. It XORs some `A[j]` into `A[i]`, with $j > i$. Doing it twice restores the initial value of `A[i]`.

We now justify the given upper-bound on the running time of this function. First, it is clear that $\texttt{hi} = \min(d-k, \ell)$. And because $\ell \leq k$, we have $\texttt{hi} \leq \min(d-k, k)$. A single iteration of the inner `while` loop does $\mathcal{O}(k+t)$ operations. Since $t \leq \ell \leq k$, this is simply $\mathcal{O}(k)$. Up to a constant factor, the total number of operations is upper-bounded by $k\binom{k}{\downarrow \min(d-k,k)}$.

We have to repeat this process on all the (non-constant) derivatives, in increasing degree order.

26b ⟨*Tweak all the derivatives* 26b⟩≡ (27a)

```
size_t i = 1;
int m[d + 2];
for (int k = 1; k < d; k++) {
        int it = litFES_combination_prepare(n, k, m);
        while (!litFES_combination_finished(n, it)) {
```

```
                tweak_derivative(n, d, A, i, k, m);
                i += 1;
                it = litFES_combination_advance(n, k, m, it);
        }
}
```

With this, we can write down the public preparation function. It does a very limited error checking on the parameters, and brutally terminates everything in case of an error.

27a ⟨*litFES Public functions* 20b⟩+≡ (25a) ◁?? 27b▷

```
void litFES_prepare(int n, int d, bool A[], bool x[],
                    struct litFES_state *s)
{
        assert(n <= BeanPolE_MAXN);
        assert(d <= BeanPolE_MAXD);
        assert(d <= n);
        ⟨Prepare the litFES_state 18a⟩
        ⟨Tweak all the derivatives 26b⟩
        for (int i = 0; i < n + 1; i++)
                x[i] = 0;
}
```

6.2 The Iteration Algorithm

We now describe how the algorithm moves to the next entry of the truth table in $\mathcal{O}(d)$ operations. This is an unrolled version of the original recursive presentation.

In the i-th iteration, d derivatives are updated, from degree $d-1$ to degree 0 (each derivative is updated using its own derivatives). Determine the location of the d rightmost "1" bits of $i+1$; for instance, write $\rho^*(i+1) = \{\ldots, b_d, \ldots, b_2, b_1\}$. These indices determine a sequence of $d+1$ monomials $m_0, m_1, \ldots, m_d$ respectively containing the variables $\emptyset, \{b_1\}, \{b_1, b2\}, \ldots, \{b_1, \ldots, b_d\}$. For $i = d-1, \ldots, 1, 0$, do `A[m[i]] ^= A[m[i + 1]]`. Finally, flip `x[b1]`.

27b ⟨*litFES Public functions* 20b⟩+≡ (25a) ◁27a 28c▷

```
void litFES_advance(int n, int d, bool A[], bool x[], struct litFES_state *s)
{
        int *f = s->f;
        int h = s->h;
        int *stack = s->stack;
        ⟨Increment i; maintain ρ(i) in k; maintain ρ*(i) in stack[0:h] 18b⟩
        ⟨Update x; return now if it is over 27c⟩
        ⟨Compute update indices; store them in r[] 28a⟩
        ⟨Update A 28b⟩
        s->h = h;
}
```

The values of x are enumerated using the reflected binary Gray code. Only one bit of x is flipped at each iteration, and its index is the least significant set bit of the new value of the iteration counter i. Therefore, we simply do:

27c ⟨*Update x;* `return` *now if it is over* 27c⟩≡ (27b)

```
x[k] ^= 1;
```

```
if (k == n)
        return;
```

As explained above, we need to perform a sequence of updates to the derivatives, of the form `A[m[i]] ^= A[m[i + 1]]`, where $m_{i+1} = m_i x_{b_{i+1}}$. The indices $b_1, \ldots, b_d$ are available in `stack[0:d]`, assuming that `stack` contains d items.

The update indices are the ranks of the monomials $m_0 \subseteq m_1 \subseteq \cdots \subseteq m_{d-1}$ defined above. litFES uses the graded lexicographic order because it allows to compute all the ranks of these nested monomials in $\mathcal{O}(d)$ operations, using the observation at the end of Subsection 5.1. The following code chunk is therefore very similar to `litFES_rank()`.

28a ⟨*Compute update indices; store them in* `r[]` 28a⟩≡ (27b)

```
size_t r[d + 1];
size_t acc = 0;
r[0] = 0;
int hh = h;                                        /* hh == min(d, h) */
if (hh > d)
        hh = d;
for (int i = 1; i <= hh; i++) {                 /* r[i] == rank of m[i] */
        int j = stack[h - i];
        acc += BeanPolE_binomial[j][i];
        r[i] = BeanPolE_N[n][i] + acc;
}
```

Once the update indices for the derivatives have been computed, actually doing the update is simple.

28b ⟨*Update A* 28b⟩≡ (27b)

```
for (int i = hh - 1; i >= 0; i-) {
        A[r[i]] ^= A[r[i + 1]];
}
```

The iteration over all entries of the truth table is complete once 2^n entries have been visited, and when it is the case x represents the integer 2^n.

28c ⟨*litFES Public functions* 20b⟩+≡ (25a) ◁27b 29b▷

```
bool litFES_finished(int n, const bool x[]) {
        return x[n];
}
```

6.3 Restoring the Initial State

The array `A` containing the coefficients of the polynomial can be modified in two places: inside `litFES_prepare` and inside `litFES_advance`. In both cases, the modifications are statements of the type `A[i] ^= A[j]`, with $i < j$. In fact, in all cases j is the rank of a monomial of strictly higher degree than that of i. Such an update can be undone by repeating it.

In the preparations, the bulk of the work is done by `tweak_derivative`, and we have seen that it is involutive. We can "un-tweak" all the derivatives by "re-tweaking" them in reverse order.

28d ⟨*Un-tweak all the derivatives* 28d⟩≡ (29b)

```
for (int k = d - 1; k >= 1; k-) {
        int m[d + 2];
        size_t i =  BeanPolE_size(n, k - 1);
        int it = litFES_combination_prepare(n, k, m);
        while (!litFES_combination_finished(n, it)) {
```

```
                tweak_derivative(n, d, A, i, k, m);
                i += 1;
                it = litFES_combination_advance(n, k, m, it);
        }
}
```

Moving on to `litFES_advance`, a moment's reflection and toying with small examples shows that if `A` denotes the array of original input values and `B` denote the end values after the 2^n iterations, then

$$B[m] = A[m] + A[m, n-1] + A[m, n-2, n-1] + ... + A[m, n-k, ..., n-2, n-1]$$

Where $n-k$ is greater than the largest variable in m and the total degree is less than or equal to d.

29a ⟨*Cancel updates done only once* 29a⟩≡ (29b)

```
size_t i = 0;
for (int k = 0; k < d; k++) {       /* correct all degree-k monomials */
        int m[d + 2];
        int it = litFES_combination_prepare(n, k, m);
        while (!litFES_combination_finished(n, it)) {
                /* largest variable of m */
                int v = (k == 0) ? -1 : m[k - 1];
                for (int l = 1; (k + l <= d) && (v + l < n); l++) {
                        /* compute the new monomial m, ..., n-2, n-1 */
                        int mx[d];
                        /* copy m */
                        for (int p = 0; p < k; p++)
                                mx[p] = m[p];
                        /* add n-1, n-2, ... */
                        for (int p = 0; p < l; p++)
                                mx[k + p] = n - l + p;
                        /* Update */
                        size_t j = litFES_rank(n, k + l, mx);
                        A[i] ^= A[j];
                }
                i += 1;
                it = litFES_combination_advance(n, k, m, it);
        }
}
```

Finally to cancel the effect of the main loop and of the preparations, we do both steps in reverse order.

29b ⟨*litFES Public functions* 20b⟩+≡ (25a) ◁28c

```
void litFES_restore(int n, int d, bool A[], struct litFES_state *s)
{
        ⟨Cancel updates done only once 29a⟩
        ⟨Un-tweak all the derivatives 28d⟩
}
```

6.4 Complexity Analysis

That the `litFES_advance` function requires at most $\mathcal{O}(d)$ operations is completely obvious from its code. The non-trivial part consists in estimating the running time of `litFES_prepare` and `litFES_restore`.

In both cases, `tweak_derivative` is invoked for each monomial m of degree k strictly less than d, and its running time is dominated by $k\binom{k}{\downarrow \min(d-k,k)}$. The total number of operations of `litFES_prepare` is thus dominated by T with

$$T := \sum_{k=0}^{d-1} k \binom{n}{k} \binom{k}{\downarrow \min(d-k,k)}$$

It was stated in [BCC+10a], and it is rather obvious, that

$$T \leq \binom{n}{\downarrow d}^2 \tag{6}$$

(indeed, all the derivatives must be evaluated and each evaluation costs as much as evaluating the input polynomial naively).

A few things that can be said outright. For starters, assume that d is fixed. Then $T = \mathcal{O}\left(n^{d-1}\right)$, while the size of the input polynomial is $\Omega\left(n^d\right)$. It follows that for fixed d and large enough n, T gets smaller than the size of the input. For small values of d, corresponding to practical attack scenarios, T is therefore under control. This corresponds to our practical experience with the algorithm and small-degree polynomials: `litFES_prepare` is never a bottleneck.

However, this may not be the case for more theoretical use cases involving higher-degree polynomials. For instance, quick numerical experiments suggest that T gets exponentially larger than the size of the input polynomial when $d = \alpha n$, for any $\alpha > 0$, as $n \to +\infty$.

What is really needed in theoretical scenarios is the assurance that the running time of `litFES_prepare` does not exceed that of the full construction of the truth table by 2^n invocations of `litFES_advance`. Towards this goal, let us first quantify what we can get from (7) using the standard bound (2):

$$T \leq 2^{2nH(d/n)} \qquad (d \leq n/2)$$

It follows that as long as $H(d/n) \leq 1/2$, we are fine. $H(x)$ is strictly increasing for $0 \leq x \leq 1/2$, and numerical root finding reveals that the unique solution of $H(x_0) = 1/2$ with $0 \leq x_0 \leq 1/2$ is $x_0 = 0.110...$. It follows that as long as $d \leq n/10$, then `litFES_advance` costs less than 2^n.

In the rest of this section, we improve upon this preliminary result. We can get a crude upper-bound on T by taking one term out of the sum:

$$T \leq d \left(\max_{k<d} \binom{k}{\downarrow \min(d-k,k)} \right) \binom{n}{\downarrow d}$$

Upper-bounding the max requires distinguishing cases, at least to get rid of the min.

- If $k \leq d/2$, then $k \leq d-k$ and $\binom{k}{\downarrow k} = 2^k$.
- If $d/2 \leq k \leq 2d/3$, then $k/2 \leq d-k \leq k$ and $2^{k-1} \leq \binom{k}{\downarrow d-k} \leq 2^k$. The trivial upper-bound is relatively tight. Here, the lower bound comes from the fact that $\binom{n}{k} = \binom{n}{n-k}$ and the fact that the series contains the first half of the terms.
- If $2d/3 \leq k$, then $d-k \leq k/2$ and we have the trivial bounds from (3).

The coefficients $\binom{k}{d-k}$, or in reverse order $\binom{d-k}{k}$, are the coefficients of the $(d+1)$-th Fibonacci polynomial. They also are the anti-diagonals of Pascal's triangle. They have been studied by Tanny and Zucker in [TZ74], where it is shown that the sequence is unimodular (increasing then decreasing) and that the maximum is reached when k is the closest integer to $d(1+\sqrt{5}/5)/2$.

This maximum is asymptotically equivalent to $5^{1/4}\phi^{d+1}/\sqrt(2\pi d)$, where $\phi = (1+\sqrt{5}/2)$ is the golden ratio [oIS02]. It follows that:

$$T \leq d^{1.5}\phi^d \binom{n}{\downarrow d} \tag{7}$$

This bound is not very tight, but it is sufficient to make progress. We prove a loose result starting from (7) and using the standard bound (2):

$$T \leq d^2 2^{n\left[\frac{d}{n}\log_2\phi + H\left(\frac{d}{n}\right)\right]}$$

Set $f(x) = x\log_2\phi + H(x)$. f is strictly increasing between 0 and 1/2, because H is. We have $f(0) = 0$ and $f(1/2) = 1.347....$ It follows that there is a unique $0 \leq x_0 \leq 1/2$ such that $f(x_0) = 1$. Numerical root finding reveals that $x_0 = 0.2567....$ This shows that $T \leq d^{1.5}2^n$ as long as $d \leq n/4$.

We conjecture, on the basis on experimental observations, that this is in fact the case as long as $d \leq n/3$. At least, we can prove that this condition is necessary. Isolating the term with $k = \lfloor 2d/3 \rfloor$ in the expression of T yields the lower-bound:

$$T \geq \frac{2d}{3}2^{2d/3}\binom{n}{\lfloor 2d/3 \rfloor} \geq \frac{2d}{3}2^{n[2d/3n + H(2d/(3n)]}$$

From there, the same kind of reasoning establishes that $1 < 2d/3n + H(2d/(3n)$ when $d \geq 0.341n$. This shows that if $d \geq 0.341n$, then $T \gg d2^n$, up to a very small constant factor.

7 Implementation of the litMOB Component

The algorithm for the in-place Moebius transform is essentially the following:

1. If there are d variables, do the classic in-place Moebius transform and stop.
2. Fix the last variable to zero.
3. Proceed recursively.
4. Flip the last variable in-place.
5. Proceed recursively.
6. Flip the last variable in-place and return.

It is implemented in `litMOB.c`:

31 ⟨*litMOB.c* 31⟩≡

```
#include <assert.h>
#include "BeanPolE.h"
⟨litMOB Internal functions 23a⟩
⟨litMOB Public functions 20c⟩
```

litMOB uses the plain lexicographic for a variety of reasons. The main one is that it allows to fix the last variable in-place, and to undo the operation. Indeed, if `A` is an array that represents the coefficients of a degree-d polynomial in n variable, then `A` has size $\binom{n}{\downarrow d}$. Fixing the last variable to zero is a no-op, because all the coefficients of monomials containing the last variable are located at the end of the array. It suffices to consider that there is one less variable and that the size of the array is reduced to $\binom{n-1}{\downarrow d}$. Fixing the last variable to one require XORing the coefficient of the monomial mx_{n-1} to the coefficient of the monomial m. This can be done in-place, and it is involutive.

Our in-place Moebius transform walks on two legs:

- The classic, in-place, Moebius transform that operates on arrays of size 2^n.
- An in-place procedure to flip the last variable in linear time.

The classic Moebius transform is well-known. It is expected that this function is only invoked on small input arrays (of size 2^d), so we do not try to optimize its memory access pattern.

32a ⟨*litMOB Internal functions* 23a⟩+≡ (31) ◁23b 32b▷

```
/*
 * Compute the Moebius transform of A[0:2**n].  This is involutive.
 * This function runs in time O(n * 2**n).
 */
static void litMOB_moebius(int n, bool A[])
{
        size_t S = 1;
        size_t N = ((size_t) 1) « n;
        for (size_t i = 0; i < (size_t) n; i++) {
                for (size_t p = 0; p < N; p += 2 * S) {
                        for (size_t j = 0; j < S; j++)
                                A[p + S + j] ^= A[p + j];
                }
                S += S;
        }
}
```

The next function extends `litMOB_combination_advance`: it additionnally provides the value of $\mu_{\downarrow d'}(m)$ for a given $d' \geq d$. In other terms, as the monomials are enumerated, their rank among all monomials of a different degree d' is computed.

32b ⟨*litMOB Internal functions* 23a⟩+≡ (31) ◁32a 33a▷

```
/*
 * Advance m[] to the next monomial of degree at most d in lexicographic
 * order.  Compute its rank in lexicographic order among monomials of
 * degree <= dprime (with dprime >= d).  This function runs in constant
 * amortized time.
 *
 * On input:
 *   k and rank respectively describe the degree and the rank of the
 *   current monomial in lexicographic order among all monomials of
 *   degree at most dprime.
 *
 * At exit:
 *   The return value is the degree of the next monomial (the new value
 *   of k).  The next monomial can be found in m[d-k:d].  rank is
 *   updated to reflect the change.
 */
static int litMOB_combination_advance2(int d, int k, int m[], int dprime,
                                       size_t *rank)
{
        /* Can we add a zero to the left? */
        if (k != d && m[d - k] != 0) {
                k += 1;
                m[d - k] = 0;
                *rank += BeanPolE_N[0][dprime-k+2];
```

```
                return k;
        }
        /* Erase leftmost digits that cannot be incremented */
        while (k > 1 && m[d-k] + 1 == m[d-k+1]) {
                *rank -= BeanPolE_N[m[d - k]][dprime-k+2];
                k -= 1;
        }
        /* Increment leftmost digit */
        *rank -=BeanPolE_N[m[d - k]][dprime-k+2];
        m[d - k] += 1;
        *rank += BeanPolE_N[m[d - k]][dprime-k+2];
        return k;
}
```

The fact that this function correctly maintains $\mu_{\downarrow d'}(m)$ follows easily from (4) and (5). It is clear that its asymptotic complexity os the same as that of `litMOB_combination_advance`, namely, constant amortized time.

This enumeration routine enables us to write the other pillar, a function that flips the last variable of a polynomial in-place, in time linear in the number of potentially modified coefficients. This is how the "implementation detail" discussed in Subsection 1.7 can be dealt with. Computing the update indices `i` and `k` actually requires most of the work. Flipping the last variable x_{n-1} is done as follows: for each monomial m degree $d-1$ in $x_0, \ldots, x_{n-2}$; determine the ranks of m and mx_{n-1}; XOR the coefficient of mx_{n-1} to that of m.

33a ⟨*litMOB Internal functions* 23a⟩+≡ (31) ◁32b

```
/*
 * On input A[] contains the coefficients of f(x0, ..., xn).  On output,
 * A[] is overwritten with coefficients of f(x0, ..., xn + 1).
 * This function runs in time linear in the number of monomials in (n-1)
 * variables of degree at most (d-1).
 */
static void litMOB_flip(int n, int d, bool A[])
{
        int m[d + 1];
        int k = litMOB_combination_prepare(d - 1, m);
        size_t i = 0;                                /* i == rank of m */
        size_t lo = BeanPolE_size(n, d);            /* rank of x_{n-1} */
        size_t hi = BeanPolE_size(n + 1, d);            /* rank of x_n */
        for (size_t j = lo; j < hi; j++) {   /* j == rank of mx_{n-1} */
                A[i] ^= A[j];
                k = litMOB_combination_advance2(d - 1, k, m, d, &i);
        }
}
```

The public function implement the recursive strategy described at the opening of this section, but the explicit recursion is removed and replaced with the implicit maintenance of a stack. The stack of recursive calls is in fact represented by the current value of `x`. The recursion stops after fixing $n-d$ variables. If `x[i] == 0`, then x_i has been fixed to zero (first recursive call, step 3 above), otherwise x_i has been fixed to one (second recursive call, step 5 above).

33b ⟨*litMOB Public functions* 20c⟩+≡ (31) ◁23c 34a▷

```
void litMOB_prepare(int n, int d, bool A[], bool x[])
{
```

```
        assert(n <= BeanPolE_MAXN);         /* minimal error-checking */
        assert(d <= BeanPolE_MAXD);
        assert(d <= n);
        for (int i = 0; i < n + 1; i++)            /* fix x[] to zero */
                x[i] = 0;
        litMOB_moebius(d, A);            /* switch to the truth table */
}
```

To advance to the next chunk of the truth table, we first re-run the Moebius transform; this converts back `A[0:2**d]` to the coefficients of the polynomial. Then we flip the right variables in x, reflecting the change in `A`. Finally, we re-run the Moebius transform to get the truth table.

34a ⟨*litMOB Public functions* 20c⟩+≡ (31) ◁33b 34b▷

```
void litMOB_advance(int n, int d, bool A[], bool x[])
{
        litMOB_moebius(d, A);     /* switch back to the coefficients */
        int i = d;                       /* unwind the recursion stack */
        while ((i < n) && (x[i] == 1)) {
                x[i] ^= 1;
                litMOB_flip(i, d, A);
                i += 1;
        }
        x[i] ^= 1;                           /* flip the last variable */
        if (i == n)
                return;
        litMOB_flip(i, d, A);
        litMOB_moebius(d, A);            /* switch to the truth table */
}
```

Detecting the end of the iteration is straightforward.

34b ⟨*litMOB Public functions* 20c⟩+≡ (31) ◁34a

```
bool litMOB_finished(int n, const bool x[])
{
        return x[n];
}
```

7.1 Complexity Analysis

It remains to determine the complexity of this procedure. The classic Moebius transform is invoked 2^{n-d+1} times on arrays of size 2^d, so the total number of operations this represents is $\mathcal{O}(d2^n)$. We now claim that the total time spent flipping variables is also $\mathcal{O}(d2^n)$. This implies the announced amortized complexity of `litMOB_advance`.

The $(d+i)$-th variable is flipped $2^{n-d-i-1}$ times. The total time spent in variable flipping is then (up to a constant factor)

$$T = \sum_{i=d}^{n-1} 2^{n-i-1} \binom{i-1}{\downarrow d-1}$$

Suppose that $d \leq n/2$. Under this assumption, using the trivial bound (3) shows that the time spent flipping variables is upper-bounded by

$$T \leq d2^{n-1} \sum_{i=d}^{n-1} 2^{-i} \binom{i-1}{d-1}$$

Because $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$, it follows that:

$$T \leq d^2 2^{n-1} \sum_{i=d}^{n-1} \frac{2^{-i}}{i} \binom{i}{d} \tag{8}$$

It then follows from lemma 1 that

$$\sum_{i=d}^{n-1} \frac{2^{-i}}{i} \binom{i}{d} \leq \frac{1}{d} \sum_{i=d}^{n-1} 2^{-i} \binom{i}{d} \leq \frac{2}{d}$$

Combining this with (8) finally shows that the total time spent flipping variable is $\mathcal{O}(d2^n)$.

8 Extensions and Future Work

This concluding section discusses how the code given in this article could be extended, and pinpoints some research perspectives.

It is not difficult to use another type `T` for polynomials entries. For instance, an actual bit field could be used in place of an array of `bool` (some `C` compiler, including `gcc`, use a full byte to store a `bool`, which wastes 7 out of 8 bits). Another relevant modification would be to replace the `bool` type by a w-bit wide integer type. This enables the algorithm to operate in "Single Instruction Multiple Data" mode and process w polynomials simultaneously.

The only way in which the library accesses the coefficients of the polynomial is by doing "update" operations of the form `A[i] ^= A[j]` where j is the rank of a monomial of strictly higher degree. These updates are scattered all over the code, but they would be easy to modify.

The in-place Moebius transform emits the truth table in chunks of size 2^d. It would not be difficult to modify the code to obtain it in chunks of size 2^k, with $k \neq d$. If $k > d$, then it would need to be out-of-place, as in the original presentation. This could potentially be more practical for some use cases. The runtime would increase to $\mathcal{O}(k2^n)$.

While the FES algorithm is intrinsically sequential, the Moebius transform offers a potential for parallelization inside the `litMOB_flip` function, that flips the last variable in linear time. It could also be possible to "backport" the idea to use a Gray code in the Moebius transform. It could lead to a constant speed-up, at the expense of not visiting the truth table in lexicographic order.

With Boolean polynomials, evaluation and interpolation are very similar, and sometimes they coincide: the classic Moebius transform does both. Adapting our in-place Moebius transform to interpolate a degree-d polynomial seems relatively straightforward. Turning the FES algorithm into an interpolation algorithm seems interesting.

This also opens up an interesting algorithmic perspective: a degree-d polynomial can be interpolated from $\binom{n}{\downarrow d}$ evaluations, for instance with its value on all monomials of degree at most d. Designing a fast procedure to convert these $\binom{n}{\downarrow d}$ evaluations to the $\binom{n}{\downarrow d}$ coefficients of the polynomial would be interesting.

In the reverse direction, evaluating a degree-d polynomial on all bit strings of Hamming weight at most d would be relevant. In [Din21a], Dinur suggests to use the FES algorithm for this purpose, on the basis that there exist "monotonic" Gray codes that enumerate all bit strings by increasing Hamming weight, while flipping one variable at a time.

Lastly, improving the complexity analysis of the setup phase of the FES algorithm given in Subsection 6.4 seems interesting.

Acknowledgments

Many thanks to Itai Dinur and Hiroki Furue for thought-provoking discussions. Gaetan Leurent and Christina Boura suggested many references.

We acknowledge financial support from the French Agence Nationale de la Recherche under project "GORILLA" (ANR-20-CE39-0002) and "PostCryptum" (ANR20-ASTR-0011).

References

[BBLP22] Augustin Bariant, Clémence Bouvier, Gaëtan Leurent, and Léo Perrin. Algebraic attacks against some arithmetization-oriented primitives. *IACR Trans. Symmetric Cryptol.*, 2022(3):73–101, 2022.

[BCC+10a] Charles Bouillaguet, Hsieh-Chung Chen, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Adi Shamir, and Bo-Yin Yang. Fast exhaustive search for polynomial systems in $\mathbb{F}_2$. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 2010.

[BCC+10b] Charles Bouillaguet, Chen-Mou Cheng, Tony (Tung) Chou, Ruben Niederhagen, Adi Shamir, and Bo-Yin Yang. Fast exhaustive search for polynomial systems in $\mathbb{F}_2$. Cryptology ePrint Archive, Paper 2010/313, 2010. https://eprint.iacr.org/2010/313.

[BCC+13] Charles Bouillaguet, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, and Bo-Yin Yang. Fast Exhaustive Search for Quadratic Systems in $\mathbb{F}_2$ on FPGAs. In *Selected Areas in Cryptography*, volume 8282 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2013. https://eprint.iacr.org/2013/436.pdf.

[BD09] Michael Brickenstein and Alexander Dreyer. Polybori: A framework for gröbner-basis computations with boolean polynomials. *Journal of Symbolic Computation*, 44(9):1326 – 1345, 2009. Effective Methods in Algebraic Geometry.

[BDL+21] Christof Beierle, Patrick Derbez, Gregor Leander, Gaëtan Leurent, Håvard Raddum, Yann Rotella, David Rupprecht, and Lukas Stennes. Cryptanalysis of the GPRS encryption algorithms GEA-1 and GEA-2. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II*, volume 12697 of *Lecture Notes in Computer Science*, pages 155–183. Springer, 2021.

[BDT22] Charles Bouillaguet, Claire Delaplace, and Monika Trimoska. A simple deterministic algorithm for systems of quadratic polynomials over f_2. In Karl Bringmann and Timothy Chan, editors, *5th Symposium on Simplicity in Algorithms, SOSA@SODA 2022, Virtual Conference, January 10-11, 2022*, pages 285–296. SIAM, 2022.

[BER76] James R. Bitner, Gideon Ehrlich, and Edward M. Reingold. Efficient generation of the binary reflected gray code and its applications. *Commun. ACM*, 19(9):517–521, 1976.

[Beu22] Ward Beullens. Breaking rainbow takes a weekend on a laptop. *IACR Cryptol. ePrint Arch.*, page 214, 2022.

[BKW19] Andreas Björklund, Petteri Kaski, and Ryan Williams. Solving systems of polynomial equations over GF(2) by a parity-counting self-reduction. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 26:1–26:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[Bou11] Charles Bouillaguet. *Algorithms for some hard problems and cryptographic attacks against specific cryptographic primitives. (Études d'hypothèses algorithmiques et attaques de primitives cryptographiques).* PhD thesis, Paris Diderot University, France, 2011.

[BP17] Ward Beullens and Bart Preneel. Field lifting for smaller UOV public keys. In Arpita Patra and Nigel P. Smart, editors, *Progress in Cryptology - INDOCRYPT 2017 - 18th International Conference on Cryptology in India, Chennai, India, December 10-13, 2017, Proceedings*, volume 10698 of *Lecture Notes in Computer Science*, pages 227–246. Springer, 2017.

[BS84] László Babai and Endre Szemerédi. On the complexity of matrix group problems I. In *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*, pages 229–240. IEEE Computer Society, 1984.

[CCNY12] Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, and Bo-Yin Yang. Solving quadratic equations with XL on parallel architectures. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 356–373. Springer, 2012.

[CFMR+17] Antoine Casanova, Jean-Charles Faugère, Gilles Macario-Rat, Jacques Patarin, Ludovic Perret, and Jocelyn Ryckeghem. GeMSS: A Great Multivariate Short Signature. Research report, UPMC - Paris 6 Sorbonne Universités ; INRIA Paris Research Centre, MAMBA Team, F-75012, Paris, France ; LIP6 - Laboratoire d'Informatique de Paris 6, December 2017.

[CHR+16] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. From 5-pass *MQ*-based identification to *MQ*-based signatures. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II*, volume 10032 of *Lecture Notes in Computer Science*, pages 135–165, 2016.

[CKPS00] Nicolas T. Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In Bart Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2000.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[DDVY21] Jintai Ding, Joshua Deaton, Vishakha, and Bo-Yin Yang. The nested subset differential attack - A practical direct attack against LUOV which forges a signature within 210 minutes. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 329–347. Springer, 2021.

[Din21a] Itai Dinur. Cryptanalytic applications of the polynomial method for solving multivariate equation systems over GF(2). In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 374–403. Springer, 2021.

[Din21b] Itai Dinur. Improved algorithms for solving polynomial systems over GF(2) by multiple parity-counting. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2550–2564. SIAM, 2021.

[DLR16] Sébastien Duval, Virginie Lallemand, and Yann Rotella. Cryptanalysis of the FLIP family of stream ciphers. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 457–475. Springer, 2016.

[DRS20] Christoph Dobraunig, Yann Rotella, and Jan Schoone. Algebraic and higher-order differential cryptanalysis of pyjamask-96. *IACR Trans. Symmetric Cryptol.*, 2020(1):289–312, 2020.

[DS05] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In John Ioannidis, Angelos D. Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security, Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005, Proceedings*, volume 3531 of *Lecture Notes in Computer Science*, pages 164–175, 2005.

[DS11] Itai Dinur and Adi Shamir. An improved algebraic attack on hamsi-256. In Antoine Joux, editor, *Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers*, volume 6733 of *Lecture Notes in Computer Science*, pages 88–106. Springer, 2011.

[Dvo90] S. Dvořák. Correspondance. *Comput. J.*, 33(2):188, 1990.

[Ehr73] Gideon Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. ACM*, 20(3):500–513, 1973.

[Fau99] Jean-Charles Faugère. A new efficient algorithm for computing grobner bases (f4). *Journal of Pure and Applied Algebra*, 139(1-3):61–68, 1999.

[Fau02] Jean Charles Faugère. A new efficient algorithm for computing gröbner bases without reduction to zero (f5). In *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, ISSAC '02, page 75–83, New York, NY, USA, 2002. Association for Computing Machinery.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[FW94] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[HST+21] Kai Hu, Siwei Sun, Yosuke Todo, Meiqin Wang, and Qingju Wang. Massive superpoly recovery with nested monomial predictions. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part I*, volume 13090 of *Lecture Notes in Computer Science*, pages 392–421. Springer, 2021.

[Ja22] *Jargon File 4.4.7*, 2022. http://www.catb.org/~esr/jargon.

[Jou09] Antoine Joux. *Algorithmic cryptanalysis*. CRC Press, 2009.

[JV17] Antoine Joux and Vanessa Vitse. A Crossbred Algorithm for Solving Boolean Polynomial Systems. In *NuTMiC*, volume 10737 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2017. https://eprint.iacr.org/2017/372.pdf.

[Knu92] Donald E. Knuth. *Literate programming*, volume 27 of *CSLI lecture notes series*. Center for the Study of Language and Information, 1992.

[Knu14] Donald Ervin Knuth. *The art of computer programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley, 2014.

[LMSI22] Fukang Liu, Willi Meier, Santanu Sarkar, and Takanori Isobe. New low-memory algebraic attacks on lowmc in the picnic setting. *IACR Trans. Symmetric Cryptol.*, 2022(3):102–122, 2022.

[LPT+17] Daniel Lokshtanov, Ramamohan Paturi, Suguru Tamaki, R. Ryan Williams, and Huacheng Yu. Beating brute force for systems of polynomial equations over finite fields. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2190–2202. SIAM, 2017.

[Mif63] Charles J. Mifsud. Algorithm 154: combination in lexicographical order. *Commun. ACM*, 6(3):103, 1963.

[oIS02] The On-Line Encyclopedia of Integer Sequences. Sequence A073028, 2002. published electronically at https://oeis.org/A073028.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.

[Rus03] Frank Ruskey. Combinatorial generation, 2003. unpublished book, available online.

[Sha49] Claude E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28:656–715, 1949.

[The13] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 5.7)*, 2013. `https://www.sagemath.org`.

[TZ74] S.M. Tanny and M. Zuker. On a unimodal sequence of binomial coefficients. *Discrete Mathematics*, 9(1):79–89, 1974.

[YC04] Bo-Yin Yang and Jiun-Ming Chen. Theoretical analysis of XL over small fields. In Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan, editors, *Information Security and Privacy: 9th Australasian Conference, ACISP 2004, Sydney, Australia, July 13-15, 2004. Proceedings*, volume 3108 of *Lecture Notes in Computer Science*, pages 277–288. Springer, 2004.

[YDH+15] Takanori Yasuda, Xavier Dahan, Yun-Ju Huang, Tsuyoshi Takagi, and Kouichi Sakurai. A multivariate quadratic challenge toward post-quantum generation cryptography. *ACM Commun. Comput. Algebra*, 49(3):105–107, 2015.