University of Southern Denmark

Department of Mathematics and Computer Science

Master's Thesis in Computer Science

Implementation and Evaluation of Dinur's MQ Solver Algorithm

Supervisor:
Associate Professor
Ruben Niederhagen

Author:
Mikkel Juul Vestergaard

Academic Year 2022/2023

Resumé

I 2021 præsenterede Itai Dinur en algoritme til at løse systemer af m multivariate polynomier af orden d i n variable, over legemet $\mathbb{F}_2$. Algoritmen er baseret på *polynomie metoden* fra feltet kredsløbs kompleksitet. I artiklen viste Dinur at hans algoritme slår såkaldte "brute force" algoritmer med en eksponentiel forøgelse i hastighed. Helt konkret blev algoritmen bevist til at have en tidskompleksitet på $n^2 \cdot 2^{0.815n}$ bit operationer for kvadratiske polynomier, samt et hukommelsesforbrug på $n^2 \cdot 2^{0.63n}$ bits efter optimering af hukommelsesforbruget. Kvadratiske polynomie systemer er fokuset i dette speciale.

I dette speciale testes denne algoritme i praksis og de praktiske målinger på algoritmen bliver holdt op mod teorien. Generelt fremviser dette speciale tre implementationer af algoritmen, i form af en Python/SageMath prototype, en hurtigere C implementation, samt en AVX-optimeret C implementation. Begge C implementationer kan også køres på flerkernede CPU'er. Koden kan enten køres gennem et dertilhørende script, eller ved brug af det dynamiske bibliotek der kan kompileres i projektet. Koden i dette speciale antages at blive kørt på GNU/Linux platforme og at blive kompileret med GCC. Koden til dette speciale kan findes ved:

`https://github.com/Moggel12/MQ-Solver`.

I praksis udfordrede implementationerne fra dette speciale ikke tilsvarende implementationer af konkurrende algoritmer, hverken i form af hukommelse eller tid. Dog viste det sig at optimeringer lavet, i praksis, i C implementationerne udgjorde en signifikant fordel når man sammenligner med teori-nære implementationer af algoritmen. Mere specifikt løste den hurtigste af C implementationerne et system på 48 variable og 48 polynomier på 25 timer, kørt på en maskine med 128 CPU-kerner (256 tråde) hvortil alle tråde var i brug. Algoritmen viste sig at have interessante egenskaber i praksis, til trods for at den ikke opnåede samme konkurrencedygtighed som den gør teoretisk. Særligt C implementationerne i dette speciale opnåede ikke deres fulde potentiale, så muligheder for fremtidige optimeringer bliver også fremlagt.

Specialet præsenterer også en *formentlig* ny måde at interpolere polynomier over legemet $\mathbb{F}_2$, som et alternativ til Möbius transformationen. Denne algoritme bliver ikke analyseret teoretisk, men bliver brugt af begge C implementationer i dette projekt og bliver målt i praksis.

Contents

Chapter 1

Introduction

1.1 Motivation

With the rise of modern-day technology, privacy and data security are more critical than ever. Cryptographic schemes based on conjectured hard mathematical problems are a key building block in enabling such properties for modern technologies. In [20], Peter Shor proposed an algorithm that would enable computationally efficient attacks on many currently in-use cryptographic schemes using sufficiently large quantum computers. Although Shor's algorithm posed a threat to cryptographic schemes based on number theoretic problems, like RSA and DSA, an actual sufficiently-sized quantum computer was yet to be seen.

Now, quantum computers had already been shown to theoretically be useful in non-cryptographic research areas as well, like modeling quantum properties of particles in material sciences. Therefore, researchers worldwide started working on practical implementations of such computing devices. The size of such quantum computers is popularly determined by the amount of *quantum bits*, also denoted *qubits*, present in the system. The amount of qubits present in quantum computer systems has drastically increased over the years, with IBM having tripled the number of qubits from 2021 to 2022 in its Osprey quantum processor [19]. This indicates rapid growth and a rapidly increasing threat against the trusted number-theoretic cryptosystems Shor challenged, albeit theoretically, in 1997.

With the rising interest in practical implementations of quantum computers, the National Institute of Technology and Standards (NIST, USA) called for new quantum-resistant cryptographic schemes, also called *post-quantum cryptography* or *PQC* for short, to be standardized. This process was announced in 2016 and has up to this point resulted in four submission rounds. In 2022, NIST ended the third submission round by selecting the CRYSTALS-KYBER scheme as a new standard for cryptographic key-establishment algorithms as well as FALCON, CRYSTALS-Dilithium and SPHINCS+ as new standards for digital signatures. Even though four new schemes were selected for standardization, a fourth round continues with the remaining schemes and some new ones.

Generally, NIST plans to standardize schemes based on different mathematical problems, so-called *PQC families*. The four main families are based on *coding theory*, *structured lattices*, *systems of multivariate polynomials*, and *hash-based.*

Multivariate cryptography revolves around two mathematical problems based on systems of multivariate polynomials, the MP problem, and the IP problem. The central problem is that of the MP problem [8], typically specialized to the MQ problem. The MQ problem revolves around solving multivariate *quadratic* polynomials, like

$$p(x_0, x_1, x_2) = x_0x_1 + x_1,$$

in a system of m polynomials. By "solve", the idea is to find the assignments of variables that evaluate $p_i(\mathbf{x}) = 0$, for all p_i in a system $\mathcal{P} = \{p_i(x_0, \ldots x_{n-1})\}_{i=0}^{m-1}$. Choosing *quadratic* as the degrees of the polynomials is often a matter of efficiency. The MQ problem is NP-complete, and therefore a balance of efficient implementations but hard to break is important when designing these cryptographic schemes.

In order to sufficiently choose the parameter sizes for these new MQ-based cryptographic schemes, the efficiency of solving such polynomial systems must be taken into account. Using existing and well-known solvers, the NIST candidate scheme `Rainbow` was broken by Ward Beullens in [1], which shows the necessity for these types of solvers in the cryptanalysis and parameter choice of post-quantum cryptographic schemes.

However, the goal of more efficient *MQ-solvers* is not only important for post-quantum cryptography. One umbrella term for different attacks on general cryptographic schemes is that of *algebraic cryptanalysis*. The goal of algebraic cryptanalysis is to map the cipher in question to a system of polynomials (alongside any further necessary information) which, when solved, yields the secret key (in this case for a symmetric key cipher).

Some examples of a reduction from breaking a cipher to the MQ problem are those of [18]. The first attack, due to Courtois and Pierprzyk [7], models the AES (Rijndael) cipher as a system of multivariate quadratic polynomials over the ring of integers modulo two ($\mathbb{Z}_2$). The second attack, due to Murphy and Robshaw [17], uses the idea of creating a new cipher called `BES` over an extension field of $\mathbb{Z}_2$. These two attacks do not successfully break AES, however, do potentially yield key extraction methods that are faster than a simple brute force procedure.

Other examples of algebraic cryptanalysis are general attacks like *cube attacks* [21] and importantly also attacks on cryptographic hash functions like that of [10]. The use cases for efficient MQ-solvers are therefore vast and they are set to be an important tool in the cryptanalysis of not only PQC schemes but also more traditional symmetric ciphers and cryptographic hash functions.

The work on efficient MQ-solvers is therefore an important part of standardizing parameter sizes for many cryptographic schemes. This thesis focuses on the polynomial method algorithm from Itai Dinur, in [9], with strong theoretical properties. The codebase for this thesis can be found at

`https://github.com/Moggel12/MQ-Solver.`

1.2 Alternative methods for solving multivariate systems over $\mathbb{F}_2$

This subsection introduces alternatives to the algorithm focused on in this thesis. The subsection does not describe these alternatives in-depth, however, will refer to relevant materials that do so. The section is neither an exhaustive survey of related algorithms.

Gröbner basis solvers. The concept of a Gröbner basis was introduced by Bruno Buchberger and has been at the core of many algorithms for solving multivariate systems over a finite field. Two important solvers based on the idea of Gröbner bases are F_4 and F_5 by Jean-Charles Faugère in [12] and [11]. These algorithms have proven quite useful both in practical implementations and in cryptanalysis. One problem with these algorithms is when applied to overdetermined polynomial systems, i.e. $m > n$, as they typically sequentially manipulate pairs of polynomials.

XL, eXtended Linearization. In response to a method called *relinearization*, the authors of [6] introduced XL. Although the procedure was shown to have strong theoretical properties, it primarily works well with rather over-defined systems, where $m \gg n$. The general idea of the method is to combine Gröbner bases with large linear systems of equations generated via the input polynomial system. Implementations related to this method of solving multivariate systems have seen some success in the Fukuoka MQ-challenges[1] Type-I and Type-III groups.

Fast Exhaustive Search. The FES procedure (fast exhaustive search) is explained in Section 2.4 as it is used internally in Dinur's polynomial method solver. It was introduced in [3] and has seen both CPU, GPU, and FPGA implementations. Standalone, this algorithm is a very effective tool for solving systems of multivariate polynomials over $\mathbb{F}_2[x_0, \dots x_{n-1}]$, and has held quite a few speed records for solving these systems.

Crossbred. In [15], a hybridized algorithm was introduced by Joux and Vitse which was shown, experimentally, to beat previous solve methods. The method uses a combination of manipulations of the input system via a Macaulay matrix representation alongside an exhaustive search on smaller, related, systems. In practice, the procedure holds a fair share of the top 5 performing algorithms in the Fukuoka MQ-challenges[2].

[1,2]`https://www.mqchallenge.org/`

Chapter 2

Preliminaries

2.1 The MQ problem

For completeness, this subsection defines the problem in focus in this thesis. *Solutions*, in this case, are any assignments to the input variables that evaluate to 0 on the polynomial(s) in question, i.e. $p(\hat{\mathbf{x}}) = 0$ means that $\hat{\mathbf{x}}$ is a solution to p, whereas $p(\hat{\mathbf{x}}') = 1$ is not a solution.

Definition 1 (The MQ Problem)**.** Given a system of m multivariate quadratic polynomials $\mathcal{P} = \{p_0, \ldots, p_{m-1}\}$, over the ring of polynomials in n variables $\mathbb{F}[x_0, \ldots, x_{n-1}]$, find values $\bar{\mathbf{x}} = (\bar{x}_0, \ldots, \bar{x}_{n-1})$ that satisfy

$$p_0(\bar{\mathbf{x}}) = p_1(\bar{\mathbf{x}}) = \cdots = p_{m-1}(\bar{\mathbf{x}}) = 0$$

Here, $\mathbb{F} = \mathbb{F}_q$ is a finite field of q elements. For this thesis $q = 2$, i.e. the field $\mathbb{F}_2 = \mathbb{Z}_2$.

2.2 Bit-slicing, partial evaluation and linearization

Two common ideas in the space of polynomials over $\mathbb{F}_2[x_0, \ldots x_{n-1}]$ are those of bit-slicing and partial evaluation. While bit-slicing refers to an optimization technique, partial evaluation is not in itself such. However, partial evaluation of polynomials can be used in tandem with other techniques, for better optimizations.

In this thesis, all *quadratic* polynomials given as input to solvers, sub-procedures, etc. are expected to be *linearized.* This is a rather simple idea, in which a quadratic polynomial is turned into a *multilinear* polynomial. Consider the polynomial

$$p(x_0, x_1) = 1 + x_0 + x_0 x_1 + x_1^2,$$

with two terms of quadratic *total degree.* However, notice that the two terms of total degree two are rather different. The term x_1^2 is quadratic in *one* variable, whereas $x_0 x_1$ is *linear* in each variable. Since the polynomial is defined over $\mathbb{F}_2[x_0, x_1]$, the monomial x_1^2 acts exactly like the monomial x_1. From this fact, p may as well be rewritten as

$$p(x_0, x_1) = 1 + x_0 + x_1 + x_0 x_1, \tag{2.1}$$

which is what *linearization* revolves around. Instead of including monomials of degree 2 in one variable, a procedure may as well preprocess the polynomial(s) making them *multilinear* like Eq. (2.1).

2.2.1 Partial evaluation

Consider the polynomial p, defined over $\mathbb{F}_2[x_0, x_1, x_2, x_3]$,

$$p(x_0, x_1, x_2, x_3) = 1 + x_0 + x_2 + x_0x_2 + x_1x_3 + x_2x_3.$$

This polynomial may be partially evaluated by assigning a subset of its variables a value, thereby obtaining a "new" polynomial. Now, assigning (fixing) k variables has 2^k possibilities. Say $k = 2$ for p above, one approach is to fix the latter k variables, $x_{n-k-1} \dots x_{n-1}$, i.e. for this example (x_2, x_3). This now results in

$$p(x_0, x_1, x_2, x_3) = \begin{cases} 1 + x_0 & (x_2 = 0, x_3 = 0) \\ 0 & (x_2 = 1, x_3 = 0) \\ 1 + x_0 + x_1 & (x_2 = 0, x_3 = 1) \\ 1 + x_1 & (x_2 = 1, x_3 = 1), \end{cases} \tag{2.2}$$

which may then be treated as four separate polynomials:

$$\begin{aligned} p^{(0)}(x_0, x_1) &= 1 + x_0 \\ p^{(1)}(x_0, x_1) &= 0 \\ p^{(2)}(x_0, x_1) &= 1 + x_0 + x_1 \\ p^{(3)}(x_0, x_1) &= 1 + x_1. \end{aligned}$$

Due to how these polynomials were generated, solving one of them will provide part of a solution for the original polynomial p. Using this idea, multiple independent polynomials can be manipulated (in parallel), while keeping a strong "relation" to the original polynomial. In case a solution for one of the $p^{(i)}$s above was found, simply extending its solution by the assignments of x_2 and x_3 is enough to *convert* the solution to one for p. I.e., $(x_0 = 1, x_1 = 0)$ is a solution for $p^{(0)}$, implying that $(x_0 = 1, x_1 = 0, x_2 = 0, x_3 = 0)$ is a solution for p. Assuming sufficient memory, this approach can also be used on entire systems.

2.2.2 Bit-slicing

Bit-slicing is another common optimization strategy when working with boolean functions, or in this case polynomials over $\mathbb{F}_2$. Bit-slicing is a parallelization strategy, which is fundamentally SIMD operations on 1-bit words. The bit-sliced procedure combines multiple bits into a single register in order to use the machine's native bit-wise operations like `xor`, `or`, and `and`, as SIMD instructions on this data. This approach is a cheap method for operating on multiple polynomials (potentially entire systems) at once, e.g. when evaluating some variable assignment on a system. In that case, the operations performed on the individual polynomial terms are the same and so the operations may as well be performed for each polynomial in parallel.

A small example of bit-slicing could be the following polynomials

$$p_0(x_0, x_1) = 1 + x_1 + x_0x_1$$
$$p_1(x_0, x_1) = x_0 + x_1,$$

that belong to a system of polynomials, $\mathcal{P} = \{p_0, p_1\}$. Choosing to write also the implicit coefficients of the polynomials (assuming the polynomials are *multilinear*),

$$p_0(x_0, x_1) = 1 + 0x_0 + 1x_1 + 1x_0x_1$$
$$p_1(x_0, x_1) = 0 + 1x_0 + 1x_1 + 0x_0x_1.$$

A procedure could fit either p_0 or p_1 into a register (byte-sized or more), by assigning each coefficient a bit-position. Assuming the procedure at some point seeks to do the same operation on terms of both p_0 and p_1 it might as well combine multiple terms into a single register, such that

$$[01_2, 10_2, 11_2, 01_2]$$

is how the system is represented, *bit-sliced.* Using the bit-wise operands of `xor` and `and` alongside the representation above, all polynomials of the system can be manipulated in parallel. E.g. an evaluation of both polynomials, for the assignment $(x_0 = 0, x_1 = 1)$, can be computed by

$$01_2 \oplus (00_2 \wedge 10_2) \oplus (11_2 \wedge 11_2) \oplus (00_2 \wedge 11_2 \wedge 01_2) = 10_2,$$

implying the evaluations be $p_0(0, 1) = 0$ and $p_1(0, 1) = 1$.

The combination of multiple data into one integer, or *register*, does not have to be related to polynomials either. Anytime the data can be represented like this, multiple parallel operations on said data can be done simply with bit-wise operands and integers/registers.

2.3 Polynomial method for solving MQ systems

According to [22], the polynomial method was originally a method for proving the limitations of computational devices in circuit complexity. That is, it was originally used for proving limitations on certain types of boolean circuits. Slowly, the method was adapted to algorithm design and has proven a useful tool for this. The general approach for using the polynomial method in algorithm design is to model the computational problem as a boolean circuit, converting it to a corresponding boolean polynomial (exact or probabilistic), to finally manipulate and compute using the polynomial.

In the field of cryptography, the polynomial method has seen multiple incarnations. Two prominent uses of the method have been for solving MP systems, one being that of [16] and the other being the main focus of this algorithm, [9]. Commonly, the system of polynomials $\mathcal{P} = \{p_i(x_0, \dots x_{n-1})\}_{i=0}^{m-1}$ are represented by

$$F(x_0, \dots x_{n-1}) = (1 + p_0(x_0, \dots x_{n-1}))(1 + p_1(x_0, \dots x_{n-1})) \dots (1 + p_{m-1}(x_0, \dots x_{n-1}))$$

with addition and multiplication being the respective operators of $\mathbb{F}_2$. Due to the structure of F, any solution (common zero) to $\mathcal{P}$ will evaluate to 1 on F. For larger systems, and of large degree, F can be hard to efficiently manipulate and is therefore often replaced by a polynomial, $\tilde{F}$, also called a *probabilistic polynomial.* Ensuring this probabilistic polynomial has a lower degree can help make the process more efficient, thereby trading determinism for efficiency.

The aforementioned structure using probabilistic polynomials is the basis of some of the core ideas in Dinur's polynomial method solver. The theoretical specification and properties of this algorithm are outlined in Chapter 3. For a more in-depth look at the algorithm please refer to the original paper in [9], which also lists cryptanalytic use cases for the algorithm.

2.4 Fast exhaustive search for multivariate polynomials

The fast exhaustive search procedure for polynomials over $\mathbb{F}_2$ was described and implemented in [3–5], is an important algorithm in the realm of practical MQ-solvers. This algorithm, typically denoted *FES*, is an exhaustive search algorithm for polynomial systems with coefficients in $\mathbb{F}_2$ in n variables and m polynomials of degree d. FES seeks to minimize the operations needed when computing all solutions of a system of multivariate polynomials. The algorithm has proven practical for real-world purposes as it was implemented on multiple occasions, with good use of the parallelization resources present in modern computers. The algorithm computes all common zeroes in maximally $2d \cdot \log_2 n \cdot 2^n$ bit operations for systems of degree-d polynomials and is a core element in Dinur's polynomial-method algorithm from [9].

To give an intuition of the algorithm, consider the polynomial

$$p(x_0, x_1, x_2) = 1 + x_0 + x_2 + x_0x_2 + x_1x_2.$$

A simple way to start an exhaustive search of solutions for p is by evaluating $p(0, 0, 0) = 1$, as this is simply the constant term of p. From this point, instead of fully re-evaluating p on a new input, assigning $x_0 = 1$ and keeping the previous assignments of x_2 and x_3 means that only the terms affected by x_0 need to be re-evaluated. Therefore, evaluating the next assignment boils down to

$$p(1, 0, 0) = p(0, 0, 0) + (1 + (1 \cdot 0)) = 1 + (1 + 0) = 0.$$

Now, observe how the computation above is similar to

$$p(0, 0, 0) + \frac{\partial p}{\partial x_0}(1, 0, 0),$$

Index	Binary	Gray
0	000_2	000_2
1	001_2	001_2
2	010_2	011_2
3	011_2	010_2
4	100_2	110_2
5	101_2	111_2
6	110_2	101_2
7	111_2	100_2

Table 2.1: 3-bit Gray codes and their corresponding binary numerals.

where $\frac{\partial p}{\partial x_0} = 1 + x_2$. Not counting how the derivative itself is obtained, evaluating $p(1, 0, 0)$ with this method is cheaper than fully re-evaluating it term-by-term.

Using the approach just described, alongside an integer counter for enumerating the variable assignments, the subsequent evaluation would be $p(0, 1, 0)$. Stepping from $p(1, 0, 0)$ to $p(0, 1, 0)$ multiple bits changed, hence multiple assignments changed (in this case those for x_0 and x_1). For larger iteration counts, up to n variables may change assignment between evaluations, hinting at the need for an *alternative* method of enumeration. The ideal case for the derivative approach is to have only *one* variable change between assignments. Enumerating all 2^n assignments in such a way can be done through Gray code order, i.e. $000_2, 001_2, 011_2, 010_2, 110_2, 111_2, \ldots$ and so on.

Finally, earlier it was hinted that computing these derivatives needs to be done efficiently. By storing computed evaluations and derivatives each time they are re-computed, future evaluations and derivatives simply need to do a lookup on the previous value. This way, first-order derivatives may be computed by adding to their previous *evaluation* the appropriate second-order derivative, depending on what variables changed between the current and previous evaluation of this first-order derivative. Of course, these principles can be generalized quite well to degree d polynomials, stopping this *recursive* idea at dth-order derivative.

2.4.1 Gray codes

An essential part of the innards of FES is that of *Gray codes* or *reflected binary codes.* This is fundamentally an ordering of the binary numbers. In this ordering, any two consecutive numbers will differ in only *one* bit. An example of this is the binary numeral of the decimal 3, using three bits, being 011_2 whereas its corresponding Gray code is 010_2. The consecutive value, decimal 4, has the binary numeral 100_2 and Gray code 110_2. These codes have various properties making them applicable in error correction, position encoders, and more. The type of Gray code used in the FES procedure from [3–5] is not the only one, as multiple other types with different additional properties exist. For an example of the Gray code ordering

Algorithm 1: EVAL(p, n)

```
Input: $p$, $n$
Result: List of common zeroes of $p$
1  $state \leftarrow$ INIT($p$, $n$)
2  if $state.y = 0$ then
3      Add $state.y$ to list of common zeroes
4  end
5  while $state.i < 2^n$ do
6      STEP($state$)
7      if $state.y = 0$ then
8          Add $state.y$ to list of common zeroes
9      end
10 end
11 return List of common zeroes
```

Figure 2.1: Pseudo-code for the total evaluation procedure of FES (quadratic polynomials)

used in FES see Table 2.1.

To construct the sequence of all n-bit binary reflected Gray codes, a recursive formulation can be used. The idea is, in each recursive step, to reflect→append→prepend. Starting with the sequence $0_2, 1_2$, one first *reflects* the sequence (resulting in $1_2, 0_2$) and *appends* it to the original sequence. Lastly, one prepends 0 to entries of the first half of this new sequence and 1 on the latter half. The sequence is now $00_2, 01_2, 11_2, 10_2$. These steps can then be repeated until the codes are formed of n bits.

Using the approach above, a "closed formula" for computing the ith codeword can be derived. For the sake of brevity, the connection will not be described more than by noting its existence. The formula for computing the ith codeword is

$$g_i = i \oplus (i \gg 1)$$

where $\gg$ denotes a logical right-shift operation in the binary representation of i and $\oplus$ denotes the bit-wise *xor* operation between two binary numbers.

2.4.2 Exhaustive Search using Gray Codes

Definition 2. Let $b_\alpha(i)$ denote the kth least significant bit in the binary representation of the decimal i. If i has hamming weight less than k, $b_\alpha(i) = -1$.

Definition 3 (Derivatives)**.** Let $\{\mathbf{e}_0, \ldots, \mathbf{e}_{n-1}\}$ denote the canonical basis over the vector-space $(\mathbb{F}_2)^n$. The derivative of a polynomial, p, in the ring $\mathbb{F}_2[x_0, \ldots, x_{n-1}]$ w.r.t. the jth variable is $\frac{\partial p}{\partial x_j} : \mathbf{x} \mapsto p(\mathbf{x} + \mathbf{e}_j) + p(\mathbf{x})$. This is also called the *finite difference* in the x_j dimension of the multivariate polynomial p. Recall that the + operator above is over $\mathbb{F}_2$, acting like an exclusive-or operation.

To minimize the number of operations needed between iterations in an exhaustive search procedure, the authors of [3] suggest looking at inputs in Gray code order. Examining inputs in Gray code order allows for efficient use of partial derivatives for computing the evaluation of one input based on the evaluation of the previous input. Inspecting Definition 3 reveals the foundation of this idea. In each iteration of the exhaustive search procedure $p(\mathbf{x}_i)$ needs to be evaluated. Using the Gray code approach, only one variable in the input changes so $\mathbf{x}_i$ and $\mathbf{x}_{i-1}$ differ in only the jth variable, meaning $p(\mathbf{x}_{i-1} + \mathbf{e}_j) = p(\mathbf{x}_i)$. From Definition 3,

$$p(\mathbf{x} + \mathbf{e}_j) = p(\mathbf{x}) + \frac{\partial p}{\partial x_j}(\mathbf{x})$$

which implies that the difference between two evaluations in Gray code order is $\frac{\partial p}{\partial x_j}(\mathbf{x}_{i-1}) = \frac{\partial p}{\partial x_j}(\mathbf{x}_i)$, and was proven in [5]. Therefore, storing $p(\mathbf{x}_{i-1})$ and adding $\frac{\partial p}{\partial x_j}(\mathbf{x}_{i-1})$ is sufficient for computing the next evaluation in a Gray-code ordered input sequence.

In other terms, let $i = 0, \dots 2^n - 1$ denote the iteration count for the FES procedure or the current index into the Gray code sequence of n-bit codewords. Between two consecutive steps of FES, say $i = 10$ and $i = 11$, the Gray codes $g_{10} = 1111_2$ and $g_{11} = 1110_2$ differ in only the least significant bit. Letting $\mathbf{x}_{10}$ and $\mathbf{x}_{11}$ be vector forms of g_{10} and g_{11}, respectively, the difference between $p(\mathbf{x}_{10})$ and $p(\mathbf{x}_{11})$ is exactly $\frac{\partial p}{\partial x_0}(\mathbf{x}_{11})$. In this example, since x_0 was the only variable that changed, the partial derivative w.r.t. x_0 represents the only parts of p that change between evaluations of $\mathbf{x}_{10}$ and $\mathbf{x}_{11}$.

Now, using the idea of derivatives will reduce the evaluation of degree d polynomials to that of evaluating a degree $d - 1$ polynomial (i.e. a first-order partial derivative). However, stopping here leaves what [3] denotes as the *folklore differential technique*. Consequently, the original authors devised the FES algorithm by (amongst other things) recursively applying this derivative idea. This means that between $i = 10$ and $i = 11$, the algorithm stores the latest $\frac{\partial p}{\partial x_j}(\mathbf{x})$ that has been computed and ensures to update it by recursively looking at the only variable, x_l, that changed since last time x_j toggled. This means that $\frac{\partial p}{\partial x_j}(\mathbf{x})$ may be computed by adding $\frac{\partial^2 p}{\partial x_j \partial x_l}(\mathbf{x})$ to the previous evaluation of $\frac{\partial p}{\partial x_j}$. For quadratic polynomials, this second derivative would be a constant (stored in a lookup table) whereas running FES on systems of degree d means recursing d times.

Letting $\mathbf{v}_i$ be the vector-form of the counter i (non-Gray code), the FES procedure looks not only for $b_1(\mathbf{v}_i)$ (Definition 2) but $b_\alpha(\mathbf{v}_i)$ for $\alpha = 1, \dots, d$. For the quadratic case in Algorithm 3 the procedures BIT_1 and BIT_2 represent $b_1(\mathbf{v}_i)$ and $b_2(\mathbf{v}_i)$ with `state.i` representing $\mathbf{v}_i$. The fact that the bits that are changed can be found through the (non-Gray code) counter can be derived through the construction of n-bit sequences of binary reflected Gray codes, or see the proof in [5]. The pseudo-code for the previously described process resides in Algorithm 3, showing both the storage of first and second derivatives as well as the computation of `state.y` which corresponds to $p(\mathbf{x}_i)$. Clearly, this idea shows how storage is one of the weaknesses of FES, especially for polynomials of degree $d > 2$.

Due to this structure, FES is required to have initialized these derivative values for whenever it encounters the *first toggle* of each bit, i.e. when `state.i + 1` sets a bit in a

```
Algorithm 2: INIT($p$, $n$)
   Input: $p$, $n$
 1 State $state$
 2 $state.i \leftarrow 0$
 3 $state.y \leftarrow p$.constant_coefficient()
 4 foreach $k = 1, \dots n-1$ do
 5     foreach $j = 0, \dots k-1$ do
 6         $s.d''[k, j] \leftarrow p$.monomial_coefficient($k, j$)
 7     end
 8 end
 9 $s.d'[0] \leftarrow p$.monomial_coefficient(0)
10 foreach $\alpha = 1, \dots n-1$ do
11     $s.d'[\alpha] \leftarrow s.d''[\alpha, \alpha-1] \oplus p$.monomial_coefficient($\alpha$)
12 end
13 return $state$
```

Figure 2.2: Pseudo-code for the initialization procedure of FES (quadratic polynomials)

position that so far has been untouched. Therefore, some time is spent pre-evaluating these derivative values directly. For the quadratic case, the pre-evaluation is done in Algorithm 2, lines 10-13, equivalently the value stored is $\frac{\partial p}{\partial x_\alpha \partial x_{\alpha-1}} + a_\alpha$ for a_α being the coefficient to the monomial x_α. An example where this is necessary is when evaluating $p(1, 1, 0, 0)$, at which point no previous $\frac{\partial p}{\partial x_1}$ evaluations exist (if not initialized). This derivative would need to be initialized to $\frac{\partial p}{\partial x_1}(\mathbf{e}_0) = \frac{\partial p}{\partial x_1 \partial x_0}$, as is shown in [5]. This last example assumes p is a quadratic polynomial.

To see that the recursive procedure above may be further optimized, one may observe that running such a procedure on a single polynomial, p, of degree d yields a complexity of $O(d \cdot 2^n)$ and consuming $O(n^d)$ bits of memory. These facts were proven in [3]. This paper further introduces smaller optimizations to the theoretical construction of the algorithm, such as removing computations of a `state.x` variable, which in Algorithm 1, 2 and 3 have already been applied.

As is also stated in [3], this initial FES version may be parallelized quite nicely, due to the evaluation procedure doing computations independently from the coefficients of the polynomials. This means that running an instance per $p_i \in \mathcal{P} = \{p_0, \dots p_{m-1}\}$ is possible, where each instance is essentially running on independent data. Extending the values (e.g. `state.y`) of Algorithm 1, 2, and 3, to be bit-vectors instead of a singular bit then yields a bit-sliced version that may use bit-wise instructions of `xor` and `and`, potentially computing a full system in one pass (in parallel). For Algorithm 2, Algorithm 1 and Algorithm 3, the pseudo-code only represents FES on single polynomials, but it should still be clear how to parallelize using bit-slicing and inputting entire systems $\mathcal{P}$ (with appropriate method calls) instead of single polynomials p.

```
Algorithm 3: STEP(state)
  Input: state
1 $state.i \leftarrow state.i + 1$
2 $\alpha_1 \leftarrow \mathrm{BIT}_1(state.i)$
3 $\alpha_2 \leftarrow \mathrm{BIT}_2(state.i)$
4 if $\alpha_2$ exists in $state.i$ then
5     $s.d'[\alpha_1] \leftarrow s.d'[\alpha_1] \oplus s.d''[\alpha_1, \alpha_2]$
6 end
7 $s.y \leftarrow s.y \oplus s.d'[\alpha_1]$
```

Figure 2.3: Pseudo-code for the procedure handling derivatives and evaluations in FES (quadratic polynomials).

2.4.3 Partial evaluation and FES

Two important ideas for obtaining a sufficient parallelization of the combined procedure (in practice) from Section 2.4.2; using bit-vectors for collecting bit operations of multiple polynomials in the system as well as *partial evaluation*. By fixing k variables, say x_{n-k} to x_{n-1}, 2^k new systems may be obtained in which each $p_i \in \mathcal{P}$ is partially evaluated on some corresponding permutation of k bits. Partial evaluation is explained in more detail in Section 2.2.1.

The approach used by the authors of [3, 4] was therefore to select w polynomials, fixing k variables and producing 2^k smaller systems of w polynomials which are searched using the recursive procedure described in Section 2.4.2. Any common zero of these systems is then checked against the remaining $m - w$ polynomials. This approach achieves the proclaimed complexity of $2d \cdot \log_2 n \cdot 2^n$, as proved in [3], which consequently is the one Dinur requires for his polynomial-method algorithm of [9].

2.4.4 Generalized FES

The ideas behind a generalized FES approach are mostly similar to what has already been described. The idea is still to use derivatives in order to minimize the operations needed to compute new evaluations of the polynomial(s). However, since input polynomials may now be of degree $d > 2$, the procedure has to use the appropriate high-order derivatives.

Consider the polynomial

$$p(x_0, x_1, x_2) = 1 + x_0 + x_0x_1 + x_0x_1x_2$$

using the FES approach detailed in Section 2.4, it is clear that

$$\frac{\partial^2 p}{\partial x_0 \partial x_1} = 1 + x_2.$$

Second-order partial derivatives are no longer constants, instead, the procedure must use third-order partial derivatives for polynomials like p above, or d-order partial derivatives for degree d polynomials in general. Therefore, the derivatives must now be stored in more and/or larger tables, as the procedure has to store up to d-order partial derivatives. These d-order partial derivatives act like constants, just like second-order partial derivatives did for quadratic polynomials. All other partial derivatives of order $< d$ are to be computed similarly to the first-order partial derivatives of Algorithm 3.

Consider Algorithm 4; the algorithm is an extension of Algorithm 1, that now computes the zeros of any polynomial of degree d, using the ideas discussed above. The dictionary D stores the derivative values of monomials just like $s.d'$ and $s.d''$ did in Algorithm 3 and Algorithm 2. However, in this approach, the algorithm stores all derivatives in a single dictionary. For the sake of simplicity, any lookup on an entry in D that has not been initialized is defaulted to return 0. Further, monomials can be represented as lists of the indices of 1-bits in an n-length bitstring, representing which of the n variables are *active* in the monomial. An example could be the monomial $x_0 x_2 x_3$ that may be represented by the bitstring (or bit-vector) 1101_2. The key zero, $D[0]$, corresponds to $s.y$ in Algorithm 1.

The `DERIVATIVE_INIT()` procedure in the pseudo-code initializes the derivative table entries to their initial values. For the quadratic case (Algorithm 2), this was a rather simple task of initializing the second-order derivative table simply as appropriate coefficients, followed by initializing the first-order derivative table to

$$\frac{\partial p}{\partial x_\alpha}(\mathbf{x}_{2^\alpha}) = \frac{\partial^2 p}{\partial x_\alpha \partial x_{\alpha-1}} + c_\alpha$$

where c_α is the coefficient to the monomial x_α. Notice, the notation from Section 2.4.2, $\mathbf{x}_{2^\alpha}$, is used to denote the Gray code value of 2^α as a vector in $\mathbb{F}_2^n$. In the general case, the goal is the same, however, for a potentially larger degree. This means that the general approach is to initialize entries to

$$\frac{\partial^j p}{\partial x_{\alpha_1} \dots \partial x_{\alpha_j}}(\mathbf{x}_{2^{\alpha_1}+\dots+2^{\alpha_j}})$$

with $\alpha_1 \dots, \alpha_j$ being the contents of the bit-vector representation of monomials described earlier. Again, formal proofs for why these initializations are used can be found in [5].

The early parts of the for-loop at Line 8 is the generalized version of the stepping procedure, Algorithm 3. Instead of only computing the positions of the two least significant one bits, for counter value i, the procedure instead computes the positions of the least d 1-bits. The amount of bit positions stored depends on whether or not the counter value i has $\leq d$ bits, if more the procedure only chooses the first d bits. The *Depth* variable in the pseudo-code represents the number of bits chosen, or how *deep* the "recursion" should go. Inspecting Line 12, it should be clear that new derivative values are computed in a recursive manner, where low-order derivatives are computed by adding the high-order derivatives, i.e.

$$\frac{\partial^{j-1} p}{\partial x_{\alpha_0} \dots \partial x_{\alpha_{j-2}}}(\mathbf{x}_i) = Q_{x_{\alpha_0} \dots x_{\alpha_{j-1}}} + \frac{\partial^j p}{\partial x_{\alpha_0} \dots \partial x_{\alpha_{j-1}}}(\mathbf{x}_i). \tag{2.3}$$

Here, $Q_{x_{\alpha_0} \dots x_{\alpha_{j-1}}}$ is the previous evaluation of $\frac{\partial^{j-1} p}{\partial x_{\alpha_0} \dots \partial x_{\alpha_{j-2}}}$, i.e. the entry stored in the derivative table. It should be noted here that recursively applying the equation above will resolve when reaching the base case of $\partial^0 p = p$, being the polynomial p itself.

Now, since the procedure stores the evaluation of $p(\mathbf{x}_i)$ in $D[0]$, the only missing part is to check if the evaluation is a common zero. This takes place at Line 14 and onwards.

```
**Algorithm 4:** GENERALIZED_FES($p$, $n$, $d$)
   **Input:** A polynomial $p$ alongside its number of variables $n$ and degree $d$.
   **Result:** All zeros of the polynomial $p$.
 1 $Solutions \leftarrow []$
 2 $k \leftarrow 0$
 3 $D \leftarrow$ DICT(default: 0)
 4 $D[0] \leftarrow p$.constant_coefficient()
 5 foreach $Mon$ in $p$.monomials() do
 6     $D[Mon.\text{bits}()] \leftarrow$ DERIVATIVE_INIT($Mon$)
 7 end
 8 foreach $i = 0, \dots 2^n - 1$ do
 9     $Depth \leftarrow \min(\text{HAMMING\_WEIGHT}(i), d)$
10     $\alpha \leftarrow \text{BITS}(i, Depth)$
11     foreach $j = Depth \dots, 1$ do
12         $D[\alpha_{0 \dots j-1}] \leftarrow D[\alpha_{0 \dots j-1}]] \oplus D[\alpha_{0 \dots j}]$
13     end
14     if $D[0] = 0$ then
15         $Solutions[k] \leftarrow \text{GRAY}(i)$
16         $k{+}{+}$
17     end
18 end
19 return Solutions
```

Figure 2.4: A generalized FES procedure for degree d polynomials.

2.5 Polynomial interpolation

Polynomial interpolation is quite an important concept in terms of Dinur's polynomial-method algorithm. The Möbius transform, as described in [14], enables obtaining the algebraic normal form (ANF) of a boolean function using its truth table as input. The algebraic normal form of a boolean function p on n variables is written

$$p(x_0, \dots x_{n-1}) = \bigoplus_{(a_0, \dots a_{n-1}) \in \mathbb{F}_2^n} g(a_0, \dots a_{n-1}) \prod_i x_i^{a_i}$$

where g is the Möbius transform. The Möbius transform g is a boolean function itself and may therefore be implemented using bit operations. The ANF of p may be represented with

Algorithm 5: MOB_TRANSFORM(S, n)

```
Input: The truth table $S$ of the boolean function $p$ on $n$ variables, with $2^n$ entries.
Result: The Möbius transform of $p$, contained in the original list of $S$.
foreach $i = 0, \dots n-1$ do
    $Sz \leftarrow 2^i$
    $Pos \leftarrow 0$
    while $Pos < 2^n$ do
        foreach $j = 0, \dots Sz-1$ do
            $S[Pos+Sz+j] \leftarrow S[Pos+j] \oplus S[Pos+Sz+j]$
        end
        $Pos \leftarrow Pos + 2 \cdot Sz$
    end
end
```

Figure 2.5: Pseudo-code of an implementation of the Möbius transform.

a bit vector of length 2^n. A closer inspection of the relationship between g and p further reveals that the Möbius transform can be computed using the formula

$$p(x_0, \dots x_{n-1}) = p^{(0)}(x_0, \dots x_{n-2}) \oplus p^{(1)}(x_0, \dots x_{n-2}) \cdot x_{n-1}$$

where $p^{(0)}$ and $p^{(1)}$ are defined as

$$\begin{aligned} p^{(0)}(x_0, \dots x_{n-2}) &= p(x_0, \dots x_{n-2}, 0), \\ p^{(1)}(x_0, \dots x_{n-2}) &= p(x_0, \dots x_{n-2}, 0) \oplus p(x_0, \dots x_{n-2}, 1). \end{aligned}$$

Now, a nice property of the Möbius transform is that it is an involution, and so it computes both the ANF of a boolean function/polynomial p via its truth table but may also use the polynomial as input and obtain its truth table, i.e. a full evaluation on its input-space.

There are different ways of computing the Möbius transform in practice. One method is the *in-place* implementation, in which the input truth table is overwritten in place. This variant is described Fig. 2.5 and an example implementation may be found in [14], alongside a more detailed description of the Möbius transform and related transforms. It should also be noted that the algorithm in Fig. 2.5 is consistent with the involution property of the Möbius transform.

Finally, using the Möbius transform for fully evaluating a polynomial may be a viable option, comparing it against naively evaluating the polynomial on all inputs one by one. The works of [10] show how one may approach such use of the Möbius transform, using the constructs above, and argues for its complexity.

2.5.1 Transformations on sparse data

When interpolating polynomials, not all 2^n assignments of the n variables are needed. In [9] it was noted that the Möbius transform algorithm may be adapted to work with

$$\sum_{i=0}^{d} \binom{n}{i}$$

evaluations in the input array, for a degree $d \leq n$ polynomial of n variables. For this to work, only evaluations of low hamming weight input vectors ($\leq d$) are used, instead of a full evaluation. It was also argued in [9] that such an adaptation would yield a time complexity of $\leq n \cdot \sum_{i=0}^{d} \binom{n}{i}$ bit operations.

Chapter 3

Dinur's polynomial-method solver

A specific instance of the polynomial method for solving multivariate quadratic polynomial systems (see Section 2.3) is the algorithm of [9], due to Dinur. As the title of the thesis states, this will be the algorithm in focus. In [16] and [22], alternative polynomial method algorithms are described. These algorithms served as inspiration for Dinur's solver.

3.1 Notation

For the following sub-sections to make sense, some notation is due, all of which is borrowed from [9].

- Let W^n_w be the set $\{\mathbf{x} \in \{0,1\}^n \mid HW(\mathbf{x}) \leq w\}$, where $HW(\mathbf{x})$ is the hamming weight of a vector $\mathbf{x}$.
- Let $\binom{n}{\downarrow w} = \sum_{i=0}^{w} \binom{n}{i}$. This is also the size of the set W^n_w.
- The subscript $z_i \leftarrow 0$ implies that a polynomial $p_{z_i \leftarrow 0}(\mathbf{y}, \mathbf{z})$ has it's ith z-variable ($z_i \in \mathbf{z}$) set to zero.

3.2 Complexities

With the concrete complexities of the algorithms in [16, 22] being larger than 2^n, a concretely efficient algorithm for cryptographic purposes was yet to be seen before [9]. In 2021, Dinur formulated a polynomial-method algorithm to be applicable in cryptography in general, and specifically for cryptanalytic purposes. This meant that the non-asymptotic complexities ought to be good even for very large problem sizes, due to the natural parameter sizes in cryptography. The asymptotic complexities of the formerly mentioned algorithms of [16, 22] may therefore be better, while in a non-asymptotic context not yielding the exponential speedup over exhaustive search as advertised. The algorithm provided in [9] therefore has the interesting property of yielding exponential speedup over exhaustive search, even for very large problem sizes. In this vein, the algorithms of [16, 22] have been revealed to have a concrete complexity larger than 2^n for cryptography-relevant parameters.

From analysis, the algorithm in [9] is bound to

$$n^2 \cdot 2^{0.815n}$$

bit operations for systems of quadratic polynomials, and

$$n^2 \cdot 2^{(1-\frac{1}{2.7d})n}$$

for systems with degree $d > 2$ polynomials. This thesis will focus on the quadratic case, as this is currently the most relevant variant for cryptography. As a cryptanalytic tool, the algorithm was estimated to reduce the security margins of cryptographic schemes like HFE and UOV, however, some MQ-based schemes have resisted attacks using this algorithm. One downside to polynomial-method algorithms in general is memory usage. The spatial complexity of this algorithm is therefore also quite vast and was shown to be reducible to around

$$n^2 \cdot 2^{0.63n}$$

bits for quadratic polynomials systems. These complexities were proven in [9] as well.

3.3 The algorithm

Definition 4 (Isolated Solutions)**.** Let $\hat{\mathbf{x}} = (\hat{\mathbf{y}}, \hat{\mathbf{z}})$ be a solution to the polynomial system $\mathcal{P}$, where $\hat{\mathbf{x}}$ is split into two parts via the variable partition $(\hat{\mathbf{y}}, \hat{\mathbf{z}})$. The solution, $\hat{\mathbf{x}}$, is called *isolated* if for any $\hat{\mathbf{z}}' \neq \hat{\mathbf{z}}$, $(\hat{\mathbf{y}}, \hat{\mathbf{z}}')$ is not a solution to $\mathcal{P}$.

At its core, the algorithm from [9] is quite simple. The essential idea is to use smaller systems of polynomials to look for solutions. These smaller systems, of course, need a certain structure to provide the most relevant solutions for the remainder of the algorithm. To avoid brute-forcing these smaller systems, the idea is to divide the variables $\mathbf{x} = (x_0, \ldots x_{n-1})$ into two parts, allowing for obtaining and iterating *isolated solutions* (Definition 4). The isolated solutions are then later used to obtain actual solutions for the system $\mathcal{P}$. The details of Dinur's polynomial-method solver can be seen in Algorithm 6, with sub-procedures in Algorithm 7 and Algorithm 8.

For clarity, let p be a polynomial over $\mathbb{F}_2[x_0, x_1, x_2]$. Now, define a *partitioning* of the variables $\mathbf{x} = (\mathbf{y}, \mathbf{z}) = (y_0, y_1, z_0)$, such that $x_0 = y_0, x_1 = y_1, x_2 = z_0$. Instead of searching for solutions in the space of all three variables, Dinur's solver searches for *isolated solutions*, using the partitioning above (Definition 4), meaning that only assignments for the $\mathbf{y}$ variables need to be searched. Let $\hat{\mathbf{x}} = (1, 1, 0)$ be a solution to p. This solution is isolated only if $p(1, 1, 1) = 1$, as this is not a solution to p (see Section 2.1). It may still be the case that assignments like $\hat{\mathbf{x}}' = (0, 1, 0)$ is a solution, as the $\mathbf{y}$ variables are different in $\hat{\mathbf{x}}$ from $\hat{\mathbf{x}}'$. Using certain constructs, the algorithm may then *recover* the $\mathbf{z}$ bits of an isolated solution, thereby obtaining all n variables assignments of $\hat{\mathbf{x}} = (x_0, \ldots x_{n-1})$.

Solving systems using the polynomial method. Inspecting Algorithm 6, one of the first things to be done is preprocessing. This step is rather simple, as it essentially involves linearizing (see Section 2.2.1) any quadratic terms in the polynomials $p_i \in \mathcal{P}$. This linearization removes any quadratic term and adds to the polynomial a corresponding linear term in the same variable. This way, the FES procedure (Line 1 in Algorithm 8) is fed a system in the correct format.

After preprocessing, and initializing ℓ and the *PotentialSolutions* list, Algorithm 6 goes on to the vital part: Computing and checking potential solutions. The main loop of the algorithm starts by generating a *uniformly random full rank (rank ℓ) binary matrix*, $A^{(k)}$, for generating the new system $\tilde{\mathcal{P}}_k$. Requiring the matrix $A^{(k)}$ to be full rank was merely done in [9] for ease of analysis. Given the large degrees of identifying polynomials for larger polynomial systems (see Section 2.3), the approach in polynomial-methods is typically to obtain a *probabilistic polynomial*, $\tilde{F}$, of a similar system. Creating the system $\tilde{\mathcal{P}}_k$ in the way described in Algorithm 6 ensures that any assignment, $\hat{\mathbf{x}}$, such that $F(\hat{\mathbf{x}}) = 1$ also has $\tilde{F}(\hat{\mathbf{x}}) = 1$. Recall from Section 2.3 that $F(\hat{\mathbf{x}}) = 1$ states that $\hat{\mathbf{x}}$ is a solution to $\mathcal{P}$. Even though F and $\tilde{F}$ agree on solutions to $\mathcal{P}$, the case where $F(\hat{\mathbf{x}}) = 0$ imposes $Pr[\tilde{F}(\hat{\mathbf{x}}) = 0] \geq 1 - 2^{-\ell}$, as proven in [9]. For this reason, $\tilde{F}(\hat{\mathbf{x}}) = 1$ does *not* guarantee that $F(\hat{\mathbf{x}}) = 1$, i.e. a solution to $\tilde{\mathcal{P}}_k$ is not guaranteed to be one for $\mathcal{P}$. Additionally, F is of degree $d \cdot m$ and $\tilde{F}$ is of degree $d \cdot \ell$, where $\ell < m$.

Searching for solutions is initially handled by searching for isolated solutions of $\tilde{\mathcal{P}}_k$ instead of $\mathcal{P}$. As it is not guaranteed that a solution to $\tilde{\mathcal{P}}_k$ is also a solution to $\mathcal{P}$, the procedure stores any isolated solutions found as *potential solutions.* These potential solutions are computed by the sub-procedure `OUTPUT_POTENTIALS()` through multiple rounds, with each round generating a new matrix $A^{(k)}$ and new random "sub"-system, $\tilde{\mathcal{P}}_k$. The `OUTPUT_POTENTIALS()` procedure can be found in Algorithm 7. The potential solutions are computed by partitioning the n variables using the parameter $n_1 < n$. Here, $\mathbf{x} = (\mathbf{y}, \mathbf{z})$ is the partitioning where the first $n - n_1$ variables are the $\mathbf{y}$ variables, and the latter n_1 variables are $\mathbf{z}$. Of course, this process of searching only for *isolated* solutions can drastically compress the search space from the 2^n possible assignments. Additionally, the idea behind using $\tilde{\mathcal{P}}_k$ is to get a faster interpolation and summation process in Algorithm 7, as $\tilde{F}$ is of degree $\leq d \cdot \ell < d \cdot m$. Of course, as already stated, the tradeoff is that potential solutions are not guaranteed to be solutions for $\mathcal{P}$. Specifically, by partitioning using $n_1 = \ell - 1$, an isolated solution $\hat{\mathbf{x}} = (\hat{\mathbf{y}}, \hat{\mathbf{z}})$ to $\mathcal{P}$ imposes that $\hat{\mathbf{x}}$ is an isolated solution to $\tilde{\mathcal{P}}_k$ with probability $\geq 1 - 2^{n_1 - \ell} = \frac{1}{2}$ (proven in [9]). This idea of searching for isolated solutions was not a novel idea introduced by Dinur, however, the alternative algorithms that use isolated solutions all searched for them on the identifying polynomial, F, instead of $\tilde{F}$.

As potential solutions are stored in each round, in order to check if one is a *candidate solution* the procedure compares newly found potential solutions with those found in previous rounds. The solutions found in the current round are stored in the `CurrPotentialSolutions` list, and later added to the history of previous potential solutions in *PotentialSolutions.* Any potential solution $(\hat{\mathbf{y}}, \hat{\mathbf{z}})$ found in more than one round is said to be a *candidate.* Once a candidate solution has been found, the procedure tests the solution by evaluating it on the polynomial system. Since evaluating the system $\mathcal{P}$ on an assignment (Line 15 in Algorithm 6) is an expensive operation, the idea of distinguishing *candidate* solutions from *potential* solutions is used to minimize the concrete complexity in this area. In [9] it is also argued that having a candidate (i.e. a potential solution found more than once) be *incorrect* is unlikely.

```
Algorithm 6: SOLVE($\mathcal{P}$, $m$, $n$, $n_1$)
   Input: $\mathcal{P}$: $\{p_j(\mathbf{x})\}_{i=0}^{m-1}$, $m$: Integer, $n$: Integer, $n_1$: Integer
   Result: A solution to the system $\mathcal{P}$
 1 PREPROCESS($\mathcal{P}$)
 2 $\ell \leftarrow n_1 + 1$
 3 $PotentialSolutions \leftarrow []$
 4 foreach $k = 0, \dots$ do
 5     $A^{(k)} \leftarrow$ MATRIX($l$, $m$)
 6     $\tilde{\mathcal{P}}_k \leftarrow \{\sum_{j=0}^{m-1} A_{i,j}^{(k)} \cdot p_j(\mathbf{x})\}_{i=0}^{\ell-1}$
 7     $w \leftarrow (\sum_{i=0}^{\ell-1} \tilde{\mathcal{P}}_k.\mathrm{degrees}()[i]) - n_1$
 8     $CurrPotentialSolutions \leftarrow$ OUTPUT_POTENTIALS($\tilde{\mathcal{P}}_k$, $n$, $n1$, $w$)
 9     $PotentialSolutions[k] \leftarrow CurrPotentialSolutions$
10     foreach $\hat{y} \in \{0,1\}^{n-n1}$ do
11         if $CurrPotentialSolutions[\hat{y}][0] = 1$ then
12             foreach $k_1 = 0, \dots k-1$ do
13                 if $CurrPotentialSolutions[\hat{y}] = PotentialSolutions[k_1][\hat{y}]$ then
14                     $sol \leftarrow \hat{y} \parallel CurrPotentialSolutions[\hat{y}]$
15                     if TEST_SOLUTION($\mathcal{P}$, $sol$) then
16                         return $sol$
17                     end
18                 end
19             end
20         end
21     end
22 end
```

Figure 3.1: The top-level procedure of Dinur's polynomial-method algorithm.

Outputting potential solutions. As already mentioned, the algorithm seeks *potential* solutions obtained from smaller systems $\tilde{\mathcal{P}}_k$ (for some $k = 0, 1, \dots$). In any iteration of the main loop of Algorithm 6, the `OUTPUT_POTENTIALS()` function computes potential solutions through the polynomials U_i, for $i = 0, \dots n_1$. These polynomials are constructed as such:

$$U_0(\mathbf{y}) = \sum_{\hat{\mathbf{z}} \in \{0,1\}^{n_1}} \tilde{F}(\mathbf{y}, \hat{\mathbf{z}}),$$

and

$$U_i(\mathbf{y}) = \sum_{\hat{\mathbf{z}} \in \{0,1\}^{n_1 - 1}} \tilde{F}_{z_{i-1} \leftarrow 0}(\mathbf{y}, \hat{\mathbf{z}})$$

for $i = 1, \dots n_1$. As is described in [9], constructing polynomials this way, allows for enumerating the isolated solutions, partitioned by $(\mathbf{y}, \mathbf{z})$ and in this case n_1. That is, for each input $\hat{\mathbf{y}}$ the remaining n_1 bits of a solution may be computed using these sums, assuming $\hat{\mathbf{y}}$ belongs to an isolated solution. In his paper, Dinur proves that if $(\hat{\mathbf{y}}, \hat{\mathbf{z}})$ is an isolated solution to $\tilde{\mathcal{P}}_k$, then $U_0(\hat{\mathbf{y}}) = 1$ and $U_i(\hat{\mathbf{y}}) = z_{i-1} + 1$, for $i = 1, \dots, n_1$. Therefore, by computing the sums (or parities) U_i for $i = 0, \dots n_1$, the latter n_1 bits of an isolated solution can be easily recovered, assuming $U_0(\hat{\mathbf{y}}) = 1$. Going through solutions and recovering the z_{i-1} bits occurs in the final loop of Algorithm 7 (Line 11).

Looking at how these polynomials/sums are constructed it is clear that they require quite a large amount of computation, should they be searched exhaustively, as they each require an exponential amount of evaluations of the polynomial $\tilde{F}$. To avoid this, the algorithm instead uses the `COMPUTE_U_VALUES()` procedure to compute interpolation points. The procedure stores these interpolation points in the V and ZV arrays, in Line 1, with V being the values used for interpolating U_0 and ZV being the values used for interpolating U_i for $i = 1, \dots, n_1$. Now, as discussed in Section 2.5 a boolean polynomial can be interpolated from its evaluations (*truth-table*, essentially) using the Möbius transform. Instead of `COMPUTE_-U_VALUES()` having to compute entire truth-tables, the Möbius transform can be modified to enable, what is here denoted as, *sparse interpolation* of a polynomial. This modification is described in [10], and briefly in Section 2.5.1. In order to minimize the cost of sparsely computing U evaluations, Dinur proved (in [9]) that the interpolations in Line 2 and Line 4 can be performed using solutions to $\tilde{\mathcal{P}}_k$ in the set and $W^{n-n_1}_{w+1} \times \{0,1\}^{n_1}$.

Once the U polynomials have been interpolated in Line 2 and Line 4 they are exhaustively evaluated on the 2^{n-n_1} assignments of the $\mathbf{y}$ variables. At this point, instead of evaluating these polynomials directly, the Möbius transform is used by giving the U polynomials as inputs to the transform procedure. As is mentioned in Section 2.5, the Möbius transform is its own inverse and can therefore be used for both interpolation and evaluation. For this to work, the polynomials need to be stored in an array structured similarly to the truth tables. This brute force approach can be seen in Line 8.

Computing interpolation points for the U polynomials. The points needed for sparsely interpolating the U polynomials may not be chosen freely. According to the proof due to Dinur in [9], the necessary points depend on the hamming weight of the initial $n - n_1$ bits of the input to $\tilde{\mathcal{P}}_k$. Dinur essentially states that in order to interpolate U_0 for some system $\tilde{\mathcal{P}}_k$, all evaluations of $\tilde{\mathcal{P}}_k$ on $(\hat{\mathbf{y}}, \hat{\mathbf{z}})$ are needed where the hamming weight of the $\hat{\mathbf{y}}$ vector is less than $w = d_{\tilde{F}} - n_1$ (Line 7 in Algorithm 6). Likewise, to interpolate U_i, for

```
Algorithm 7: OUTPUT_POTENTIALS($\tilde{\mathcal{P}}_k$, $n$, $n_1$, $w$)
    Input: $\tilde{\mathcal{P}}_k$: $\{r_i(\mathbf{x})\}_{i=0}^{\ell-1}$, $n$: Integer, $n_1$: Integer, $w$: Integer
    Result: A two-dimensional list of size $2^{n-n_1} \times (n_1+1)$ containing the $z_i$ bits, $y$
            bits and $U_0(y)$ bit.
 1  $(V, ZV[0, \dots (n_1-1)]) \leftarrow$ COMPUTE_U_VALUES($\tilde{\mathcal{P}}_k$, $n$, $n_1$, $w$)
 2  $U_0 \leftarrow$ MOB_TRANSFORM($V[0 \dots |W_w^{n-n_1}| - 1]$, $n - n_1$)
 3  foreach $i = 1 \dots n_1$ do
 4      $U_i \leftarrow$ MOB_TRANSFORM($ZV[i][0, \dots, |W_{w+1}^{n-n1}| - 1]$, $n - n_1$)
 5  end
 6  $Evals[0 \dots n_1][0 \dots 2^{n-n1} - 1] \leftarrow \{0\}$
 7  foreach $i = 0 \dots n_1$ do
 8      $Evals[i][0 \dots 2^{n-n1} - 1] \leftarrow$ MOB_TRANSFORM($U_i$.as_array(), $n - n_1$)
 9  end
10  $Out[0 \dots 2^{n-n1} - 1][0 \dots n_1] \leftarrow \{0\}$
11  foreach $\hat{y} \in \{0,1\}^{n-n_1}$ do
12      if $Evals[0][\hat{y}] = 1$ then
13          $Out[\hat{y}][0] \leftarrow 1$
14          foreach $i = 1 \dots n_1$ do
15              $Out[\hat{y}][i] \leftarrow Evals[i][\hat{y}] + 1$
16          end
17      end
18  end
19  return $Out$
```

Figure 3.2: The subprocedure for retrieving candidate solutions.

$i = 1, \dots n_1$ all evaluations are needed where the hamming weight of the $\hat{\mathbf{y}}$-bits are less than $w + 1$.

To obtain evaluations of $\tilde{\mathcal{P}}_k$, usable for the interpolation in `OUTPUT_POTENTIALS()`, Dinur specifies using the FES procedure (Section 2.4), denoted `BRUTEFORCE()` at Line 1 in Algorithm 8. Using this procedure means that interpolation points can be obtained with time complexity

$$2d \cdot \log n \cdot 2^{n_1} \cdot \binom{n - n_1}{\downarrow w}.$$

However, as the FES procedure itself is not designed to iterate through sparse sets of inputs, such as $W_w^{n-n_1} \times \{0,1\}^{n_1}$, an overhead is incurred. The n_1 latter bits can be iterated using the normal Gray code traversal of FES, meaning a penalty is incurred for each 2^{n_1} iteration. A conservative estimate by Dinur confines this to an amortized penalty of $2^{-n_1} \cdot n$ over the FES procedure. As the complexities derived for Dinur's polynomial method algorithm assume cryptographically relevant parameter sizes, $2^{n_1} \gg n$ means the overhead is negligible.

Once the relevant evaluations for $\tilde{\mathcal{P}}_k$ have been computed, the procedure goes on to com-

Algorithm 8: COMPUTE_U_VALUES($\tilde{\mathcal{P}}_k$, n, n_1, w)

```
Input: $\tilde{\mathcal{P}}_k$: $\{r_i(\mathbf{x})\}_{i=0}^{\ell-1}$, $n_1$: Integer, $w$: Integer
Result: Lists $V$ and $ZV$ containing evaluations of
        $U_i(y), \forall i \in \{0, \dots n_1\}, \forall y \in \{y \mid y \in \{0,1\}^{n-n_1}, hw(y) \le w\}$
1  $Sols[0 \dots L-1] \leftarrow$ BRUTEFORCE($\tilde{\mathcal{P}}_k$, $n$, $n1$, $w+1$)
2  $V[0 \dots |W_w^{n-n1}| - 1] \leftarrow \{0\}$
3  $ZV[0 \dots n_1][0 \dots |W_{w+1}^{n-n_1}| - 1] \leftarrow \{0\}$
4  foreach $s \in Sols$ do
5      $\hat{y}, \hat{z} \leftarrow s[0 \dots n-n_1-1], s[n-n_1 \dots n-1]$
6      if HAMMING_WEIGHT($\hat{y}$) $\le w$ then
7          $idx \leftarrow$ INDEX_OF($\hat{y}$, $n-n_1$, $w$)
8          $V[idx]$++
9      end
10     foreach $i = 1 \dots n_1$ do
11         if $z_i = 0$ then
12             $idx \leftarrow$ INDEX_OF($\hat{y}$, $n-n_1$, $w+1$)
13             $ZV[i][idx]$++
14         end
15     end
16 end
17 return $V, ZV[1 \dots n_1]$
```

Figure 3.3: The subprocedure for computing interpolation points.

pute the actual interpolation points for the U polynomials. The loop at Line 4 (Algorithm 8) computes the evaluations for the U polynomials by iterating through all evaluations and filtering depending on the values of the $\hat{\mathbf{y}}$ and $\hat{\mathbf{z}}$ bits. As the U_0 polynomial can be interpolated from the evaluations of $\tilde{\mathcal{P}}_k$ in $W_w^{n-n_1} \times \{0,1\}^{n-n_1}$, Algorithm 8 simply checks whether or not the current $\hat{\mathbf{y}}$ bits have a hamming weight $\le w$, before summing and storing. Likewise, to filter out evaluations for some U_i, the procedure inspects the $\hat{\mathbf{z}}$-bits and checks whether or not $z_{i-1} = 0$ (as is necessary by the construction of U_is). Recall that as $\tilde{F}(\hat{\mathbf{x}}) = 1$ for any solution $\hat{\mathbf{x}} = (\hat{\mathbf{y}}, \hat{\mathbf{z}})$ to $\tilde{\mathcal{P}}_k$, the U polynomial evaluations are essentially parities for the number of solutions with specific *properties* (such as $HW(\hat{\mathbf{y}}) < w$) to $\tilde{\mathcal{P}}_k$.

It should be noted that the Gray code traversal of the FES sub-procedure could also be reimplemented using monotone Gray codes instead of traditional binary reflected Gray codes, as noted by Dinur in [9]. This way, the Gray code traversal is done in an *almost* increasing order according to the hamming weight, instead of often having to skip iterations because $\hat{\mathbf{y}} \in \{0,1\}^{n-n_1} \setminus W_{w+1}^{n-n_1}$

Mitigating storage problems. As is also noted by Dinur in [9], computing several rounds in parallel can help mitigate some of the storage problems of the overall solver. In Algorithm 7 exponentially sized lists/tables are computed and later stored, in order to check for *candidate* solutions across rounds. If instead the U polynomials for multiple rounds could be evaluated in parallel (or at least one after the other), their evaluations could be checked in parallel as well. There are multiple approaches for this, one of which is described in later sections and another described by Dinur in [9]. This does require certain modifications to the algorithm, like computing multiple $\tilde{\mathcal{P}}_k$ systems in advance of calling `OUTPUT_POTENTIALS()`.

Chapter 4

Extensions

The spatial complexity of Dinur's polynomial-method algorithm is one of the greatest hindrances to an implementation of the solver being viable in practice. This problem was also acknowledged by Dinur in [9] and a mitigation was introduced as the *memory efficient Möbius transform*. This section introduces an alternative to the original proposal.

4.1 Polynomial interpolation using FES

As an alternative to the Möbius transforms used in Algorithm 7, interpolation based on the same principles as the FES procedure of [3] may be used. In order to introduce this procedure recall the *generalized FES*, introduced in Section 2.4.4. The generalized approach is described in detail in [5] if more information is needed. Since this alternative FES-based approach is supposed to replace the interpolation and evaluation in the Möbius transform, the procedure must be able to handle polynomials of variable degrees.

In [9], using FES for evaluation instead of the Möbius transform was hinted at, however, such an approach was dismissed due to the initialization overhead. This overhead, however, is not present when fully evaluating polynomials using the Möbius transform. While the arguments make sense, the approach to be discussed here does not require an initialization phase, thereby making it a more viable replacement of the Möbius transforms executed in Algorithm 7.

4.1.1 FES-based interpolation and recovery.

At last, comes an introduction to more *novel* ideas; the FES-based interpolation and *recovery* of solutions. In the context of Dinur's polynomial-method solver, the most interesting of the two would be recovering solutions. For this reason, most of the focus will be on how to recover solutions of a degree d polynomial.

Interpolation using FES. Observe how derivative table entries are updated using the principles of Eq. (2.3). Reversing the computation allows for *backtracking* from an evaluation

Algorithm 9: FES_RECOVER(d, n, S)

```
Input: For some polynomial $p$, the degree $d$, amount of variables $n$, and a sparsely
       filled truth-table $S$.
Result: The full truth-table of $p$, stored in $R$.
1  $R[0 \dots 2^n - 1] \leftarrow \{0\}$
2  $D \leftarrow$ DICT(default: 0)
3  $R[0], D[0] \leftarrow S[0], S[0]$
4  foreach $i = 1 \dots, 2^n - 1$ do
5      $Depth \leftarrow \min(\text{HAMMING\_WEIGHT}(i), d)$
6      $\alpha \leftarrow \text{BITS}(i, Depth)$
7      if HAMMING_WEIGHT($i$) $> d$ then
8          foreach $j = Depth \dots, 1$ do
9              $D[\alpha_{0 \dots j-1}] \leftarrow D[\alpha_{0 \dots j-1}] \oplus D[\alpha_{0 \dots j}]$
10         end
11     else
12         $Q \leftarrow D[0]$
13         $D[0] \leftarrow S[\text{GRAY}(i)]$
14         foreach $j = 1 \dots, Depth$ do
15             if $j < Depth$ then
16                 $Tmp \leftarrow D[\alpha_{0 \dots j}]$
17             end
18             $D[\alpha_{0 \dots j}] \leftarrow D[\alpha_{0 \dots j-1}] \oplus Q$
19             if $j < Depth$ then
20                 $Q \leftarrow Tmp$
21             end
22         end
23     end
24     $R[\text{GRAY}(i)] = D[0]$
25 end
26 return $R$
```

Figure 4.1: Pseudo-code for a procedure using FES to recover all evaluations of a polynomial, given a subset of its evaluations.

to a derivative table entry, i.e. computing

$$\frac{\partial^j p}{\partial x_{\alpha_0} \dots \partial x_{\alpha_{j-1}}}(\mathbf{x}_i) = \frac{\partial^{j-1} p}{\partial x_{\alpha_0} \dots \partial x_{\alpha_{j-2}}}(\mathbf{x}_i) - Q_{x_{\alpha_0} \dots x_{\alpha_{j-1}}} \tag{4.1}$$

meaning that the high-order derivatives may be computed from the values of low-order derivatives. By unpacking or unrolling the equation above through recursively applying the equality, it becomes clear that $\partial^0 p = p$ is now the base case. A procedure where the input is the full truth table of the polynomial p could compute the derivative table entries in reverse order of what is done in Algorithm 4, using Eq. (4.1). Going through all evaluations and using the reversed equation above will result in the same state (of the derivative table) as just after the initialization phase of Algorithm 4. This idea is important, however, inverting the initialization phase is not useful for this thesis, so it will not be discussed further. Just note that if one were to invert this, the coefficients of the polynomial could be obtained directly, hence interpolating the polynomial.

Recovery of all solutions using interpolation. Going back to Algorithm 7, the idea here is to introduce an alternative way of recovering solutions. Recall that currently, the recovery approach is to interpolate the U polynomials using the Möbius transform, followed by fully evaluating all of them using a separate run of the Möbius transform. Using the ideas discussed earlier in this section, this can also be done in one pass-through of the input truth table (the same *sparse* table as given to the Möbius transform for interpolation).

As in Algorithm 7, recovering the full truth table of a boolean polynomial p, of degree d, can be done using a combination of the ideas discussed in this section. For the procedure to work, the interpolation points must be the same as for the sparse Möbius transform (see Section 2.5 and Section 3.3). This means that evaluations of $\tilde{\mathcal{P}}_k$ on inputs with hamming-weight $\leq d$ are required. Now, a hybrid approach of generalized FES and FES-based interpolation can be used to *recover* all evaluations of the U polynomials.

The hybrid approach is shown in Algorithm 9, in which the *then* part of the outer-most conditional is the evaluation part, while the *else* is the interpolation part. The evaluation part of the procedure is more-or-less the same as Line 12 in Algorithm 4. The interpolation part, on the other hand, is a more descriptive version of Eq. (4.1). Since the goal was to *only* recover the full truth table and not fully interpolate p, the algorithm may ignore inverting the initialization phase (as Line 6 in Algorithm 4, or Algorithm 2). If the goal was to also fully interpolate p, this process could be added after looping through all 2^n inputs.

Also, notice how the interpolation phase is counteracting the need for a set-up phase. Since the procedure interpolates derivative entries for low-hamming-weight inputs before the evaluation phases on large-hamming-weight inputs, there is no need for an initialization phase for the evaluation part (like Algorithm 2 or Line 6 in Algorithm 4). In essence, the interpolation correctly sets up and updates the derivative table entries from the *sparse* set of input evaluations until the evaluation phase starts taking over by updating derivatives and evaluating inputs.

For clarity, the pseudo-code in Algorithm 9 requires the input n, which is the number of variables. In the case of Dinur's solver, the number of variables is $n - n_1$, as the procedure should interpolate the U polynomials.

Chapter 5

Implementation

The accompanying git repository contains more than one implementation, or *variant*, of Dinur's original algorithm. These variants are divided into two C implementations and a prototype SageMath implementation. C function declarations can be found in the `inc/` folder and other code can be found under `src/`. These implementations, alongside everything else worked on for this thesis, can be found at

`https://github.com/Moggel12/MQ-Solver`.

For alternative implementations of some of the procedures described in Chapter 2, see [3], [4], and [1].

5.1 Platform and architecture

This section describes the platform and architecture used for building, testing, and benchmarking the programs written for this thesis. For a description of the actual implementation see the previous subsections.

The compiler and operating systems used in this thesis are GCC and GNU/Linux. Therefore, running a non-GNU/Linux operating system or compiling using a non-GCC C/C++ compiler is not guaranteed to work. For running the codebase in a containerized environment, refer to the accompanying Dockerfile. Although most modern CISC CPUs are based on 64-bit instructions, compilation targets also exist for 32, 16, and 8-bit platforms (or registers).

Some of the most important instruction sets present today, other than the common CPU instructions, are SIMD (Single Instruction Multiple Data) instruction sets. Three such instruction sets are SSE2 (Steaming SIMD Extensions), AVX (Advanced Vector Extensions), and AVX2. These are all present on modern x86 CPUs, however, some also include the AVX512 instruction set. The compilation targets in this thesis only support up to AVX/AVX2. These instruction sets allow using 128-bit and 256-bit vector registers for computation on multiple data elements in one instruction. AVX512 enables further support for 256-bit vector registers while also supporting 512-bit vector registers.

The AVX/AVX2 instructions operate on the `ymm`[0-15] 256-bit registers, while the SSE2 instructions operate on the `xmm`[0-15] 128-bit registers. These instruction sets support vector operations like loads/stores, bit-wise operations, addition, subtraction, shuffling, and much more. The data packed into a vector may be both floating point data or integer data, however, in this project only integer data is necessary. The integer sizes packed into vector registers may also vary, depending on the needs. As will become clear when discussing optimizations to the C code, this project either uses 8- or 16-bit integers. To examine the full extent of these instruction sets, as well as their latencies and throughputs, refer to the respective documentation at the CPU manufacturer (such as Intel or AMD).

The parts of the code that make use of vector instructions do not use GCCs inline assembly or separate assembly files. Instead, the programs using vector instructions make use of GCC intrinsic functions for these instruction sets, i.e. those documented in [13]. How these instructions are used can be read in Section 5.4.2.

Finally, the implementations can also be told to use most of the multithreading resources present on multicore platforms. Currently, the code only supports allocating a power-of-two amount of CPU cores, so certain CPUs may not be fully utilized. The code also does not distinguish between the *performance* and *efficiency* type cores found in certain contemporary CPUs.

5.2 SageMath code

As was implied earlier, the SageMath implementation of Dinur's algorithm works mostly as a prototype or testing ground for the C implementation. Some optimizations have been tested in this version of the code, prior to it being implemented in C, however, these optimizations worked on an algorithmic level more than on a machine level. This prototype allowed for approximating the bottleneck areas of the algorithm while essentially also working as a proof-of-concept for using Dinur's algorithm in practice. These approximations of course were rougher in some areas than others, due to the overhead imposed by SageMath and Python.

The prototype implements the three procedures described by Dinur in [9], more or less described as the pseudo-code is presented. The three main procedures described by Dinur can be found in `src/sage/dinur.sage` with some accompanying convenience and test functions. A bit-sliced version of the FES procedure, described in [3] and Section 2.4, for quadratic polynomials can be found in `src/sage/fes.sage`. This implementation is not as heavily optimized as those in [3] and [4], simply due to the SageMath-induced overhead counteracting fine-adjusted optimizations. The prototype code also introduces an FES-based recovery procedure, acting as an alternative to the Möbius Transform originally described by Dinur (see Chapter 4). The Möbius Transform was implemented in `src/sage/mob_-new.sage` and allows for a *sparse*-transform used for interpolating the U-polynomials. This implementation is rather naive as it interpolates these polynomials *symbolically* using the polynomial classes from SageMath. The choice of switching between FES-based interpolation and using the Möbius transform is a simple boolean switch in the `solve()` and `output_potentials()` functions in `src/sage/dinur.sage`.

The tests for the SageMath procedures can be found in the same file as the procedure they test. This is simply due to the prototype nature of the SageMath code, therefore the SageMath code is primarily used for verification of the C-code. As references to practical implementations of these procedures are sparse, the SageMath code was rather important as it eliminated the normal headaches of working in C and allowed for a more theory-near approach. Considering that SageMath has procedures built-in for working with polynomials, matrices and rings, it was a very important stepping stone towards a C implementation. Once the prototype was finished, implementing the C code was easier as most of the algorithmic ideas had already been exercised.

Other than the prototype code implemented in SageMath, a *front-end* was also implemented allowing for easier loading systems, generation systems, and calling of the optimized C code. For more on this, see Section 5.8.

5.2.1 Dinur's core procedures

SageMath implementation of `SOLVE()`. The top-level `solve()` procedure can be found in the `src/sage/dinur.sage` file. To test it, one may call the `test_sage_solve()` function with appropriate parameters. This implementation of Dinur's algorithm tries to mimic the pseudo-code (see Algorithm 6) closely, e.g. by using dictionaries for comparing potential solutions from the current round with those of earlier rounds. However, by close inspection, one might see that there are few differences between the implementation and the pseudo-code still. In the pseudo-code, Dinur parameterizes the variable n_1, allowing variation on how it is chosen. The SageMath implementation fixes this to

$$n_1 \approx \lceil \frac{n}{5.4} \rceil.$$

The choice of fixing n_1 to this specific value stems from Dinur's proof of the time complexity of this algorithm. Setting the parameter to approximately $\frac{n}{5.4}$ ensures that the complexity is balanced between the time evaluating the U polynomials and the time taken for computing the evaluations of $\tilde{\mathcal{P}}_k$ in the set $W_{w+1}^{n-n_1} \times \{0,1\}^{n_1}$. This can be altered in the SageMath source code itself if necessary, however, here it was kept simple.

Another part of the SageMath code that differs from the source material is its `fes_recovery` parameter. This parameter handles whether or not to use FES-based recovery, described in Section 4.1, to recover the U polynomials. The parameter is essentially a boolean switch that tells the `output_potentials()` function which implementation is needed. A look at the main loop inside the `solve()` function shows the last *major* deviance from the pseudo-code of Algorithm 6. Here, instead of allowing the algorithm to run indefinitely the length of the *history* is limited. The limit found here can be changed in `src/sage/c_config.py` and defaults to 30.

Generating the matrix $A^{(k)}$ of Algorithm 6, Algorithm 6, for constructing $\tilde{\mathcal{P}}_k$ occurs in `gen_matrix_rank_l()`. Ensuring that matrix $A^{(k)}$ has rank ℓ is a simple Las Vegas (algorithmic) approach generating new matrices until one with the needed rank is acquired.

The generation of the matrix makes use of the `rand()` function from the C standard library. The PRNG is seeded in `solve()` using the constant `RSEED`, defaulting to 42. The underlying PRNG may be changed in the `src/sage/c_config.py` file as well, however, is useful for simplifying the testing of the C implementation.

Now, the way polynomials are represented in the SageMath code is through the built-in (in SageMath) representation of boolean polynomials. As mentioned earlier, this does incur an overhead but will also simplify certain operations, such as generating the system $\tilde{\mathcal{P}}_k$:

(Listing 5.2.1) src/sage/dinur.sage

```
P_k = [sum(GF(2)(A[i][j]) * system[j] for j in range(m)) for i in
↪ range(l)]
```

which also eases the process of computing $w = d_{\tilde{\mathcal{F}}} - n_1$,

(Listing 5.2.2) src/sage/dinur.sage

```
w = sum(f.degree() for f in P_k) - n1
```

alongside evaluating the polynomials in the system on candidate solutions:

(Listing 5.2.3) src/sage/dinur.sage

```
def eval_system(system, sol):
    return not any(f(*sol) for f in system)
```

Finally, instead of going through all 2^{n-n_1} assignments for $\hat{y}$, as in Algorithm 6, the SageMath version stores solutions in `defaultdicts`. This way, the procedure may only iterate through stored $\hat{\mathbf{y}}$ values, i.e. those where $U_0(\hat{\mathbf{y}}) = 1$:

(Listing 5.2.4) src/sage/dinur.sage

```
for y_hat, potential_sol in curr_potential_sol.items(): # Iterate through
↪ solutions instead of all possible inputs
    for k1 in range(k):
        if all(potential_sol == potential_solutions[k1][y_hat]):
            sol = convert(y_hat, n - n1) + list(potential_sol[1:])
            if eval_system(system, sol):
                _time_solve_trials += time.time()
                if c_debugging:
                    return sol, (k + 1)
                return sol
            break
```

The `c_debugging` part should be ignored here. The other parts should then show a strong resemblance to Algorithm 6.

Outputting isolated solutions in reality. The function `output_potentials()` is the SageMath equivalent of Algorithm 7. To compute isolated solutions using the U polynomials, the SageMath implementation takes two approaches, as noted earlier. The `fes_recovery` parameter chooses either an FES-based interpolation and evaluation or Dinur's proposed method of using the boolean Möbius transform and using `compute_u_values()`. With Dinur's proposed method of computing isolated solutions, the code first obtains `V` and `ZV`, being a `defaultdict` and a list of `defaultdict`. Once those have been saved, the procedure goes on to interpolate the U polynomials and store them in an array, `U`:

(Listing 5.2.5) src/sage/dinur.sage

```
U.append(mob_transform(V, ring_sub.gens(), w))
for i in range(1, n1 + 1):
    U.append(mob_transform(ZV[i - 1], ring_sub.gens(), w+1))
```

using the appropriate parameters (w for U_0 and $w+1$ for the other U_is). Here, the procedure also includes `sub_ring` which is a polynomial ring with indeterminates x_0 through x_{n-n_1-1} instead of x_0 through x_{n-1}, as the U polynomials are defined over the $\mathbf{y}$ variables (the first $n-n_1$ variables). Using this approach helped simplify the prototype implementation, as the Möbius transform then could be implemented by the recursive formula (see Section 2.5). This does of course add the overhead of addition and multiplication using SageMath polynomial classes while potentially also using large amounts of stack-based memory. However, as the SageMath code acts as a prototyping platform, this is not necessary to change.

Although the traditional Möbius transform takes in either an array representing a polynomial to be evaluated or the full set of evaluations in order to interpolate a polynomial, the code called in the snippet above works as originally intended by Dinur. The transform takes in sparse sets of evaluations in order to interpolate the U_0 and U_i polynomials, with the weight values defining the recursion depth for the Möbius transform.

Note, the way that the procedure is implemented using the Möbius transform for interpolation and evaluation is not the most efficient, however, it does resemble the pseudo-code on an abstract level. Should one be interested in implementing the Möbius transform in Python or SageMath and running the code with that instead, the code may be extended via the `src/sage/mob_new.sage` file. Also, the symbolic method of creating an array representing the polynomial, then evaluating and then converting the output to an array of evaluations (seen in the snippet below)

(Listing 5.2.6) src/sage/dinur.sage

```
for i in range(n1 + 1):
    tmp = [0] * 2^(n-n1)
    for m in U[i].monomials():
        if m == 1:
          v = 0
        else:
          v = str(m)
          v = v.replace('x', '')
          v = [int(i) for i in v.split("*")]
          v = sum([2^i for i in v])
        tmp[v] = GF(2)(1)

      tmp = mob_transform(tmp, ring_sub.gens())

      evals[i] = [0] * 2^(n-n1)
      for m in tmp.monomials():
        if m == 1:
          v = 0
        else:
          v = str(m)
          v = v.replace('x', '')
          v = [int(i) for i in v.split("*")]
          v = sum([2^i for i in v])
        evals[i][v] = GF(2)(1)
```

should also see a revision if the Möbius transform is more ”properly” implemented in `src/sage/mob_new.sage`. Pseudo-code and explanation of the Möbius transform can be found in Section 2.5.

The remainder of the implementation ensures to translate evaluations of the U_i polynomials into the actual z_{i-1} bit of an isolated solution, depending on the evaluation of U_0. The code can be seen below.

(Listing 5.2.7) src/sage/dinur.sage

```
for y_hat in range(2^(n - n1)):
    if evals[0][y_hat] == 1:
        if y_hat not in out: out[y_hat] = np.full(n1 + 1, GF(2)(0))
        out[y_hat][0] = GF(2)(1)
        for i in range(1, n1 + 1):
            out[y_hat][i] = evals[i][y_hat] + 1
```

The output dictionary `out` is stored as a `defaultdict` basically due to memory concerns. This of course adds processing, but it should be clear by now that execution speed was not always a priority in the SageMath code.

If the caller alternatively sets the `fes_recovery` parameter to `True`, the algorithm uses the FES-based interpolation and evaluation, described in Section 4.1. The gist of the code when going with an FES-based interpolation is the following (ignoring the timing code, of course):

(Listing 5.2.8) src/sage/dinur.sage

```
evals = fes_recover(system, n, n1, w + 1, ring)

_time_fes_recovery += time.time()

_time_fetch_sol -= time.time()

for y_hat in range(2^(n - n1)):
    if evals[y_hat] & 1 == 1:
        if y_hat not in out: out[y_hat] = np.full(n1 + 1, GF(2)(0))

        out[y_hat][0] = GF(2)(1)
        for i in range(1, n1 + 1):
            out[y_hat][i] = GF(2)((evals[y_hat] >> i) & 1) + 1 #
            ↪  Bitsliced indexing
```

The most notable difference between the two approaches is the combination of interpolation and evaluation into one. Since both the U_0 and the U_i polynomials are interpolated in the same procedure, the *weight* or `w` parameter is set to `w + 1` to accommodate for interpolation of both the U_is and U_0. The hybrid approach is described generally in Section 4.1.1.

Computing U-polynomial interpolation points. Computing the interpolation points, used for the U-polynomials takes place in much the same way as Dinur described it in [9]. The procedure `compute_u_values()` in `src/sage/dinur.sage` handles this, and is more or less the same setup as Algorithm 8. One difference that affects performance (compared to the theoretical setup) is the use of dictionaries (`defaultdict` specifically), instead of lists, as the storage solution for the interpolation points.

(Listing 5.2.9) src/sage/dinur.sage

```
V = defaultdict(lambda: GF(2)(0))
ZV = [defaultdict(lambda: GF(2)(0)) for _ in range(n1)]
```

As already discussed, regarding `output_solutions()`, this choice was merely made due to memory concerns as the `defaultdict` serves as a way of doing lookups on non-existing keys without taking up too much valuable memory.

5.2.2 FES procedures

Bruteforce and FES: One aspect of the `compute_u_values()` process that is somewhat different from [9] is the `bruteforce()` procedure. The description in [9] leaves much to the imagination as it is only really stated that the FES procedure of [4] and [3] would be used to evaluate the sparse set of inputs, $W_{w+1}^{n-n_1} \times \{0,1\}^{n_1}$. However, Dinur does make arguments for two general approaches and their performance penalties/impacts. These alternatives are described as either simply iterating through the set $W_{w+1}^{n-n_1}$ while going through all $\{0,1\}^{n_1}$ values at each such iteration, or the use of *monotonic Gray codes*. The structure of monotonic Gray codes was briefly mentioned in Section 2.4.1. Due to simplicity and the negligible performance penalty, the choice here was to use the former approach.

The `bruteforce()` procedure can be found in `src/sage/fes.sage` and is in essence rather simple. The function simply slices the polynomial system into $1+n+\binom{n}{2}$ integers and uses the bit-sliced representation to run the FES procedure on the entire system at once. The procedure loops through a sequence of integers $i = 0, \ldots 2^{n-n_1} - 1$ skipping any value of i where the $hw(i) > d$ (hamming weight) for a parameter $d = w + 1$. For each value of i where $hw(i) \leq w + 1$, a *prefix* is computed and stored as a list of indices for the 1 bits in i.

(Listing 5.2.10) src/sage/fes.sage

```
prefix = [pos for pos, b in enumerate(reversed(bin(i)[2:])) if b == "1"]
```

This prefix represents the value of i, by storing the indices of the 1-bits, and is used for representing the FES procedure *state*.

The FES procedure was not originally intended to evaluate across a sparse set of inputs and so some modifications had to be made. First off, the FES code relies on a data class `State`:

(Listing 5.2.11) src/sage/fes.sage

```
@dataclass
class State:
    i: int
    y: int
    d1: list
    d2: list
    prefix: list
```

In many areas, this state representation is similar to that of Algorithm 1, however, it has an added `prefix` attribute. This attribute helps the state class maintain information between different calls to the `fes_eval()` procedure. The state, `s`, is updated whenever a new value of $i \in W_{n_1}^{n-n_1}$ is reached in the counter using the `update()` procedure:

(Listing 5.2.12) src/sage/fes.sage

```
s = update(s, system, n, n1, prefix)
```

This process is simply partially evaluating the polynomials of $\tilde{\mathcal{P}}_k$ on the counter value i, in the first $n - n_1$ variables. When a new counter value, $i' \in W_{w+1}^{n-n_1}$, is reached the `update()` function ensures to *turn on* or *turn off* (set variables to 1 or 0, respectively) the bits in the derivatives `s.d1` and `s.d2`, as well as the evaluation `s.y`. This process depends on which variables were changed to 1 and which were changed to 0 going from counter value i to i'. An example of this process is the following code:

(Listing 5.2.13) src/sage/fes.sage

```
for idx in off:
    for k in range(n1):
        s.d1[k] ^^= f[lex_idx(idx, k + (n - n1), n)]

    s.y ^^= f[idx + 1]
```

This snippet goes through all variables that are turned off, going from i to i', and subtracts the coefficient of the monomial $x_{idx}x_{k+n-n_1}$ from the value of $\frac{\partial f}{\partial x_{k+n-n_1}}$. Likewise, the "evaluation", or `s.y`, also takes effect from changing the assignments of the first $n - n_1$ variables. For this reason, the procedure must ensure to subtract the coefficient of the monomial x_{idx} from the evaluation `s.y`. By Adding and subtracting these coefficients, the procedure computes a partial evaluation and its effect on the derivatives as well. The benefit of doing this is that the FES initialization phase only has to run *once*, as the state may simply just be updated when reaching counter value i'.

Similarly, the `update()` function handles the other effects of changing assignments of the first $n - n_1$ variables. Variables are also handled similarly to the process just described when turned *on* between two calls to `update()`. In general, the procedure turns *on* or *off* variables in the derivative tables and evaluation `s.y` based on which monomials are affected by said variable assignment change. By then storing the `prefix` inside the state, the procedure can subsequently handle new changes without having to entirely reboot the FES procedure.

The subsequent code in `bruteforce()` may then simply call the `fes_eval()` procedure with the newly update state `s` as such:

(Listing 5.2.14) src/sage/fes.sage

```
sub_sol = fes_eval(system, n, n1, prefix, s)
```

which internally works much like the pseudo-code described in Section 2.4. The function `fes_eval()` handles two cases, depending on whether or not it is called by `bruteforce()`

or `fes_recover()`. For now, this section will focus on the code of `bruteforce()` and related parts. The function declaration looks like

```
(Listing 5.2.15) src/sage/fes.sage
def fes_eval(f, n, n1 = None, prefix=[], s = None, compute_parity=False):
```

where the `compute_parity` parameter is set to false when called from `bruteforce()`, as this then ensures that the procedure returns solutions, like traditionally done in FES. The parameters of `f`, `n` and `n1`, should be clear from Chapter 2 and Chapter 3, or [9] and [4]. The parameters of `prefix` and `s` relate to the state processing described just now.

Whenever the `fes_eval()` function is called with `s = None`, the code has to initiate a new state `s`. In the same vein as the `update()` function, the `init()` function (in `src/sage/fes.sage`) ensures to initiate the first and second derivatives according to the prefix that the system is being partially evaluated on. This leads to computations like

```
(Listing 5.2.16) src/sage/fes.sage
for idx in prefix:
    for k in range(n1):
        s.d1[k] ^^= f[lex_idx(idx, k + (n - n1), n)] # Alter this
        ↪  function in old FES

    s.y ^^= f[idx + 1]
```

where the computation is very much like the one in `update()`. Of course, other than initializing the state according to the current prefix the state also needs to be initialized in accordance to Algorithm 2.

Once the state has been initialized, the execution of `fes_eval()` follows the ideas of Algorithm 1 closely. However, the procedure essentially runs an exhaustive search on the space $\{0,1\}^{n_1}$, but solutions from $W_{w+1}^{n-n_1} \times \{0,1\}^{n_1}$ are needed. Therefore, instead of storing the value of $s.i \oplus (s.i \gg 1)$ (the Gray code value of `s.i`), the binary representation of the prefix has to be prepended to the Gray code counter.

Once all solutions have been found and stored with the currently set *prefix*, the procedure may not yet return control to `bruteforce()`. Since the `state` is re-used and updated between successive calls to `fes_eval()`, the state also has to reset certain values. That is, at the end of `fes_eval()` the procedure resets `s.d1`, `s.d2` and `s.i` to their initial values from when `fes_eval()` was initially called. The snippet here shows this process:

```
(Listing 5.2.17) src/sage/fes.sage
for i in range(n1-1):
    s.d1[i] ^^= s.d2[n1-1][i]
s.y ^^= s.d1[n1-1] ^^ s.d2[n1-1][n1-2]
```

where the code "subtracts" second derivatives added to the first derivative (during the search for solutions) for each of the first $n_1 - 1$ *unassigned* (not partially evaluated) variables. Following this, the procedure can reset `s.y`. Inspecting the xor operations of Algorithm 3, it should be clear how this process resets `s.d1` and `s.y`. Of course, the counter in `s.i` is also reset to ensure that the next run of `fes_eval()` only goes through $\{0,1\}^{n_1}$ as well.

The last part of the `bruteforce()` procedure in `src/sage/fes.sage` adds the solutions obtained by `fes_eval()` as lists of $GF(2)$ elements. Once all prefixes, or all values of $W^{n-n_1}_{w+1}$, have been processed the algorithm returns all solutions found.

FES-based recovery: The procedures STEP, BIT$_1$, and BIT$_2$ from Algorithm 3 are all implemented in a quite straightforward manner, meaning that they will not be explained here. In return, the `fes_recover()` function from `src/sage/fes_rec.sage` probably deserves some explanation. There are essentially three parts to this function; `fes_recover()` itself, `fes_eval()` with `compute_parities` set to `True`, and `part_eval()`. As the observant reader may have already noticed, the `fes_recover()` function is an implementation of the procedure described in Section 4.1 or Algorithm 9.

As is described in Chapter 4, the FES-based recovery (Algorithm 9) acts as an alternative to interpolation and evaluation using the Möbius transform. In the case of Dinur's solver, this means that it needs to fill the entries of a $(n_1 + 1) \times 2^{n-n_1}$-sized array, containing evaluations of the U polynomials. In the SageMath code, these entries are bit-sliced such that the evaluations of the $n_1 + 1$ U polynomials fit into a single integer in a 2^{n-n_1}-sized array.

For the FES-based recovery to work, the procedure requires the set $W^{n-n_1}_{w+1} \times \{0,1\}^{n_1}$ as interpolation points, just like the Möbius transform. Internally, the `fes_recover()` procedure computes these points, instead of having to pre-compute them and pass them as input to the procedure. For this reason, the `state` class (Listing 5.2.11) is used once more. The `fes_recover()` procedure internally partially evaluates the system $\tilde{\mathcal{P}}_k$, like in `bruteforce()`, before computing the interpolation points. Now, `fes_recover()` also calls `fes_eval()` to compute these interpolation points, however, the `compute_parities` parameter is set to `True` this time. Passing this parameter changes the behavior of `fes_eval()` from computing solutions of a partially evaluated system, to directly computing the summations from Algorithm 8 (like Line 4). The value returned from `fes_eval()` is now instead the evaluation of $U(\mathbf{y}_{prefix})$, with $\mathbf{y}_{prefix}$ being the vector representation of `prefix`.

Now, since the FES-based recovery needs to interpolate and evaluate polynomials of variable degree (i.e. the U polynomials), the procedure must implement less specialized code than the likes of Algorithm 1, or `fes_eval()` in `src/sage/fes.sage`. In the SageMath code, this is handled by the derivative *table* (dictionary) `d` and the array of bit-positions `alpha`. Essentially this means that `fes_recover()` internally uses two different kinds of FES; a specialized version for quadratic polynomials (computing solutions of $\tilde{\mathcal{P}}_k$), and the generalized version for interpolation and evaluation.

Fundamentally, the way the table `d` stores derivative values of varying order is by simply interpreting the counter `si` as a bitstring, and then selecting maximally the `degree` first 1-bits of this bit-string. An example of this is $si = 101101_2$ and $degree = 3$, where the index would be $si = 1101_2 = 13$. This bit-string is computed using the `bits()` function, returning an array of indices for all 1-bits in the counter `si`:

```
(Listing 5.2.18) src/sage/fes_rec.sage
alpha = bits(si)[:degree]
```

The variable `alpha` is here simply an array of these indices. This is very much just the generalized version of $\text{BIT}_1()$ and $\text{BIT}_2()$ in Algorithm 3, or simply α and BITS() in Algorithm 9. The values in this array `alpha` are then used to compute the indices into the derivative table, according to which monomial `si` represents. An example of the computation of indices into `d` is:

```
(Listing 5.2.19) src/sage/fes_rec.sage
d[sum([2^i for i in alpha[:j]])] = int(d[sum([2^i for i in alpha[:j]])])
 ↪  ^^ int(d[sum([2^i for i in alpha[:j+1]])])
```

It should further be noted that index 0 in `d` represents the evaluation of `si` on the U polynomials, just like in Algorithm 9. Also, the derivative table `d` is bit-sliced, such that each entry contains the corresponding values for each U polynomial.

Then, as the procedure seeks to evaluate the U polynomials on $\{0,1\}^{n-n_1}$, it goes through all values $i = 0, \dots 2^{n-n_1} - 1$. At each iteration, the procedure chooses to interpolate or evaluate given the current iteration count `si`. In much the same sense as interpolation using the Möbius transform on sparse inputs, the interpolation of `fes_recover` occurs whenever the counter value has $hw(i) < d$ where d is the *degree* specified through the `degree` parameter to `fes_recover`. In all other cases, the procedure conversely evaluates the polynomial, given the interpolations computed.

Focusing on the interpolation part for a bit, it can be seen that the line

```
(Listing 5.2.20) src/sage/fes_rec.sage
s, new_parities = part_eval(system, prefix, n, n1, s)
```

is rather important. The prefix, much like in `bruteforce()`, represents the input to the various U polynomials (as discussed alreay); $U(\mathbf{y}_{prefix})$. Internally, the `part_eval()` function updates the state `s` and computes *parities* through `fes_eval()`. These parities are then simply returned alongside the updated state `s`.

Now, the *parities* are nothing more than the values computed in `compute_u_values()`. An example of the parity computation in `fes_eval()` is

(Listing 5.2.21) src/sage/fes.sage

```
parities ^^= 1
z = (s.i ^^ (s.i >> 1))
for pos in range(n1):
    if z & (1 << pos) == 0:
        parities ^^= (1 << (pos + 1))
```

Again, this is bit-sliced and so each integer contains n_1+1 bits, corresponding to each of the U polynomials. Comparing the snippet above to the main loop of `compute_u_values()` (or Algorithm 8) it should be clear that these are more-or-less the same computations. Setting `compute_parities` in `fes_eval()` to `True` does not affect anything else than how solutions are handled. The procedure still resets the `state` attributes and still computes evaluations the same as when set to `False`.

Once the *parities* of a certain prefix have been computed, they may be stored in the derivative table immediately as these represent an evaluation of the U polynomials. Once these have been stored, the procedure can do the *backtracking* described in Section 4.1. This process is implemented in the following way:

(Listing 5.2.22) src/sage/fes_rec.sage

```
prev = d[0]
d[0] = new_parities


for j in range(1, len(alpha)+1):

    if j < len(alpha):
        tmp = d[sum([2^i for i in alpha[:j]])]

    d[sum([2^i for i in alpha[:j]])] = int(d[sum([2^i for i in
    ↪  alpha[:j-1]])]) ^^ int(prev)

    if j < len(alpha):
        prev = tmp
```

To add some explanation, the procedure computes the high-order derivatives using the values of low-order derivatives (and the evaluation) in the for-loop. Of course, the `prev` variable corresponds to Q in Algorithm 9.

Conversely, if the hamming weight of the counter is sufficiently high, $hw(si) > degree$, the procedure may *evaluate* instead of interpolating. This process does not need the parity computation, as the low hamming weight entries in the derivative table have already been updated by earlier interpolation steps, as explained in Section 4.1. Therefore, the code is simply going bottom-up computing the high-order derivatives first, like Line 12 (in Algorithm 4). For these computations, the `degree` first 1-bits of *si* are once more used in order to compute the derivative table entries required. The following snippet shows these computations:

```
(Listing 5.2.23) src/sage/fes_rec.sage
for j in reversed(range(0, len(alpha))):

    d[sum([2^i for i in alpha[:j]])] = int(d[sum([2^i for i in
    ↪  alpha[:j]])]) ^^ int(d[sum([2^i for i in alpha[:j+1]])])
```

Once the current iteration either finished interpolation or evaluation, the procedure stores `d[0]` in the array of evaluations:

```
(Listing 5.2.24) src/sage/fes_rec.sage
res[si ^^ (si >> 1)] = d[0]
```

Given that this approach is a variant of FES, the evaluation in `d[0]` is the result of evaluating the U polynomials on the Gray code value of `si`, and must therefore be stored accordingly in the `res` array.

As mentioned in Algorithm 9, combining interpolation and evaluation into one procedure allows for a single pass-through of the 2^{n-n_1} evaluations needed for the U polynomials. This way, the procedure does not need to store both exponentially-sized arrays for representing the U polynomials plus the *Evals* array of Algorithm 7. However, the derivative table is still quite large, and the computational needs for interpolation and evaluation are still expensive. For an evaluation of using FES-based recovery, see Chapter 6.

5.2.3 Möbius Transform and utilities

Brief note on the Möbius transform. As mentioned earlier, the Möbius transform implementation(s) can be found in `src/sage/mob_new.sage`. The procedure `mob_transform()` represents the Möbius transform, however, instead of using lists as described in Section 2.5 the representation uses the built-in SageMath representation of boolean polynomials. This oddity is the reason that Listing 5.2.6 is as comprehensive. Implementing the Möbius transform as described in Section 2.5, alongside the *sparse* Möbius transform, will make much of the code in Listing 5.2.6 redundant and more readable. By using the boolean polynomial classes of SageMath, computing the Möbius transform is performed through the recursive formula specified in Section 2.5. Implementing the sparse interpolation can be done by filtering out high-degree monomials after fully interpolating the polynomial from its sparse set of solutions when the transform is computed in this ”symbolic” way. For inspiration on implementing the Möbius transform less symbolically, implementations like that of [2] are worth a look.

Utilities. The SageMath codebase in `src/sage/` provide quite a few extra utility functions, depending on the goal. The `src/sage/utils.sage` file provides procedures for reading and writing Fukuoka MQ-challenge style files[1], bit-slicing multilinear quadratic polynomials, generating polynomial systems, fetching C functions from the shared library, and more. These utilities are used throughout the SageMath codebase and also in the `run.py` script, in order to handle smaller tasks that would not require a dedicated SageMath file. Should one seek to extend parts of the SageMath code, or the `run.py` script, this file should be kept in mind.

5.3 Core algorithms in C

In many areas, the standard C implementation takes a similar approach to program design as the SageMath counterpart (see Section 5.2). However, unlike the intent with the SageMath prototype, the purpose of the C code was to test the algorithm and see what kinds of optimizations are beneficial for it. This subsection seeks to describe the general idea behind the C implementation, drawing parallels to the similar parts in the SageMath code and describing the difference alongside their design choices. Evaluating the usefulness of these optimizations is postponed to Chapter 6. This subsection focuses on the *shared library* compile-target (see Section 5.7) as well as the *standardized* implementation, plus any utilities that overlap between codebases.

5.3.1 Solve, and other top-level procedures

The main entry point into the library is the `solve()` procedure, residing in `src/c/standard/mq.c`. This procedure for the most part acts like its SageMath counterpart. However, the `solve()` procedure expects a bit-sliced system of polynomials in its `poly_t *system` parameter. Further, the procedure expects that the monomial ordering is graded lexicographic ordering and that the polynomials in the system have been *linearized*, i.e. the system consists solely of multilinear polynomials. Examining the function declaration, the other parameters should be somewhat self-explanatory:

(Listing 5.3.1) inc/mq.h

```c
uint8_t solve(poly_t *system, unsigned int n, unsigned int m, poly_t
↪   *sol);
```

The latter parameter, `sol`, is an out-parameter containing the solution found, if any.

Now, due to the unrestricted nature of C, the initial part of the procedure allocates memory for the elements like the sub-system $\tilde{\mathcal{P}}_k$, the matrix $A^{(k)}$ and the solution history:

[1]https://www.mqchallenge.org/

(Listing 5.3.2) src/c/standard/mq.c

```c
SolutionsStruct *potential_solutions[MAX_HISTORY] = {0};
size_t hist_progress[MAX_HISTORY] = {0};

poly_t *rand_sys = malloc(amnt_sys_vars * sizeof(poly_t));

poly_t *rand_mat = malloc(l * sizeof(poly_t));
```

$\tilde{\mathcal{P}}_k$ and $A^{(k)}$ are stored in the `rand_mat` and `rand_sys` variables simply as integer arrays, and reused in each round. The solution history is saved as an array of pointers, called `potential_solutions` and its size is defined by the `MAX_HISTORY` macro. This macro defines an upper limit on how many rounds the solver may use to search for a solution and can be modified in `inc/mq_config.h`. The types of these three variables are not exactly default types in C.

One prominent type in both the vectorized and standard implementation is the `poly_t` type. This type is simply a type definition reusing different integer types in C:

(Listing 5.3.3) inc/mq_config.h

```c
typedef POLY_TYPE poly_t;
```

The `POLY_TYPE` macro is defined as a uint{8, 16, 32, 64}_t for the *standard* implementation, depending on the compile-flags (see Section 5.7). The definitions of both the `POLY_TYPE` macro and `poly_t` reside in `inc/mq_config.h`. The other important type is the `SolutionsStruct`. This struct will be discussed later.

Once sufficient memory has been allocated and relevant parameters computed (such as n_1 and ℓ) the main loop of the algorithm starts. In the SageMath version, the matrix generation and subsequent generation of the subsystem $\tilde{\mathcal{P}}_k$ could be handled by the polynomial and matrix representation built into SageMath. In the C code, these procedures have to be handled manually.

Generating the $A^{(k)}$ matrices of Algorithm 6 is done through a call to `gen_matrix()` in `src/c/utils.c`. The procedure takes as argument an array of `poly_t` to fill and the number of rows, `n_rows`, and columns, `n_columns`, for the matrix. Since the matrix is supposed to consist of elements from $\mathbb{F}_2$, each row is an integer sampled in the `gen_-row()` procedure, thereby representing the row as an integer. These integers are masked so that only the bottom `n_cloumns` bits are left. The `gen_matrix()` procedure then simply generates the full $\ell \times m$ matrix and computes the rank, by computing the entire row echelon form of the matrix and counting independent rows in the meantime. The procedure generates matrices using a Las Vegas (algorithmic) approach, by continuously generating matrices until one of rank ℓ is obtained.

Given the matrix generated consists of m bits inside a `poly_t`, the procedure for generating $\tilde{\mathcal{P}}_k$ is quite straightforward. Recall that the new system is generated by

$$\tilde{\mathcal{P}}_k = \left\{ \sum_{j=0}^{m-1} A_{i,j}^{(k)} \cdot p_j(\mathbf{x}) \right\}_{i=0}^{\ell-1} \tag{5.1}$$

The `compute_p_k()` procedure generates the new system polynomial-by-polynomial, term-by-term. Now, this procedure generates a system of multilinear polynomials, keeping it consistent with the rest of the codebase. As the polynomials are saved bit-sliced, the summation and multiplication in Eq. (5.1) is merely a matter of:

```
(Listing 5.3.4) src/c/standard/mq.c
new_sys[0] = GF2_ADD(new_sys[0], parity(GF2_MUL(old_sys[0], mat[s])) <<
↪  s);
```

Here, `GF2_ADD` is a macro for bitwise *xor* and `GF2_MUL` is a macro for bitwise *and*, these and other macros are described in Section 5.5. The variable `s` above is equivalent to the i in Eq. (5.1). The snippet above is the computation for the constant terms of $\tilde{\mathcal{P}}_k$, however, by sufficient indexing into `old_sys` and `new_sys` the same approach may be taken to compute other coefficients as well. At last, the procedure also computes the degrees of these polynomials meanwhile.

Once a new system of polynomials has been computed, the procedure goes on to compute the potential solutions for the current iteration. This is where one of the larger deviations from the SageMath code and Algorithm 6 takes place. First, the C solver solely makes use of FES-based recovery (see Section 4.1), with no alternative Möbius transform implementation. Second, the `solve()` procedure calls the `fes_recover()` procedure directly:

```
(Listing 5.3.5) src/c/standard/mq.c
error = fes_recover(rand_sys, n, n1, w + 1, curr_potentials, &len_out);
```

Since the procedure of Algorithm 7 primarily combines interpolation, evaluation, and ”translating” the recovered evaluations into potential solutions, no real benefit exists that required it to be implemented when the `fes_recover()` procedure is used. With the structure of `fes_recover()`, the evaluations can be translated and stored directly in the array of solutions *as* they are computed, instead of going through 2^{n-n_1} iterations (in `fes_recover()`) to compute all evaluations, and then subsequently iterate through all 2^{n-n_1} evaluations once more in order to translate them into potential solutions (Line 11 in Algorithm 7). If a Möbius transform procedure would have been implemented, the necessity for an implementation of Algorithm 7 would be larger. The actual implementation of `fes_-recover()` is discussed in Section 5.3.2 and Section 5.4.

Since the FES-based recovery of Section 4.1 is equivalent to the Möbius transforms (interpolation *and* evaluation) of Algorithm 7, breaking the theory is not a risk. Now, the reasoning behind not including a Möbius transform implementation was the *implementability*, so to say. The `fes_recover()` version was the more desirable choice, both in order to introduce a use-case for this new procedure and also due to its seemingly simple nature. FES implementation also seems to have been tested more in scientific literature, with examples in [4], [2], and [3]. This, however, does not mean the Möbius transform cannot be implemented nor that `fes_recover()` should be the sole choice for any other implementations of Dinur's polynomial method solver.

The potential solutions computed in `fes_recover()` are stored inside a `poly_t`. In this representation, the **y**- and **z**-bits of the potential solution are stored in the first $n - n_1$ and latter n_1 bits, respectively. Further, the **y**-bits are *not* stored as their Gray code value.

With the *solution checking* phase of the C implementation, the code once again diverges from the SageMath and reference material (Algorithm 6). Instead of storing the solutions in a dictionary or full $2^{n-n_1} \times (n_1 + 1)$ size array, only potential solutions with $U_0(\hat{\mathbf{y}}) = 1$ are stored. These are stored sequentially in an array. The storage of these lists of potential solutions is done through the struct:

```
(Listing 5.3.6) src/c/standard/mq.c
```
```c
typedef struct SolutionsStruct
{
  poly_t *solutions;
  size_t amount;
} SolutionsStruct;
```

Since the code cannot simply make a lookup into a dictionary or array using the **y**-bits as key or index, the history checking phase differs quite a bit from Algorithm 6 and Listing 5.2.4.

Due to the way that `fes_recover()` is implemented, the potential solutions are stored with their **y**-bits (recall, not Gray code) in increasing order (interpreting these bits as unsigned integers). Since the conversion to and from Gray code ordering acts like a bijective mapping, two **y** values that are equal correspondingly mean that their Gray code values are equal. This way, searching for some Gray code value in an array ordered this way may be performed by linearly comparing (non-Gray code) **y**-bits. Once some **y**-bits of a larger *magnitude* have been found, the procedure is guaranteed that the Gray code value initially searched for is not present.

Using the sorting just discussed, the procedure may simply loop through all previous lists of potential solutions, using the sorting of the **y**-bits as an indicator of whether or not a solution could still be found. That is, the procedure checks all the previous lists of potential solutions (`k1`):

```
(Listing 5.3.7) src/c/standard/mq.c
```
```c
for (unsigned int k1 = 0; k1 < k; k1++)
```

Here, each iteration then goes through the `poly_ts` stored in the history, or the `k1`th old list of potential solutions:

(Listing 5.3.8) src/c/standard/mq.c

```c
for (; hist_progress[k1] < potential_solutions[k1]->amount;
     hist_progress[k1]++)
```

until it encounters a historic solution with a larger non-Gray code $\mathbf{y}$ value;

(Listing 5.3.9) src/c/standard/mq.c

```c
if (GF2_MUL(k1_solution, INT_MASK((n - n1))) > GF2_MUL(idx_solution,
↪  INT_MASK((n - n1))))
{
  break;
}
```

or one where the historic solution and the newly found solution are identical;

(Listing 5.3.10) src/c/standard/mq.c

```c
else if (INT_EQ(k1_solution, idx_solution))
```

This ensures that the procedure checks only the strictly necessary historic solutions, linearly. The linear search could also be replaced with a binary search-like strategy. Alternatively, a dictionary could of course be implemented, however, it is unclear whether or not this would prove a dramatic benefit. Another alternative would be to allocate all $2^{n-n_1} \times (n_1 + 1)$ entries in a table as originally proposed in [9]. Of course, choosing to allocate room for all evaluations of the U polynomials will drastically limit the problem sizes that the algorithm may handle in practice (on machines without unlimited memory).

Finally, should the procedure encounter identical potential solutions, the code acts more or less as expected:

(Listing 5.3.11) src/c/standard/mq.c

```c
              poly_t gray_y = GRAY(GF2_MUL(idx_solution, INT_MASK((n -
              ↪  n1))));
              poly_t solution =
                  GF2_ADD(gray_y, GF2_MUL(idx_solution,
                  ↪  INT_LSHIFT(INT_MASK(n1), (n - n1))));

              if (!eval(system, n, solution))
              {
                *sol = solution;
```

```

                    for (unsigned int i = 0; i <= k; i++)
                    {
                      g_stored_solutions += potential_solutions[i]->amount;
                      free(potential_solutions[i]->solutions);
                      free(potential_solutions[i]);
                    }

                    free(rand_sys);
                    free(rand_mat);

                    END_BENCH(g_solve_time)

#if defined(_DEBUG)
                    solver_rounds = k + 1;
#endif

                    return 0;
                  }
```

In this snippet, the code ensures to convert the **y**-bits into its Gray code counterpart, before combining the **y**- and **z**-bits into one integer (or *candidate solution*). The candidate solution is then evaluated on $\mathcal{P}$ to check if it is an actual solution. If so, the procedure cleans up and returns with the solution stored in the out-parameter `poly_t *sol`. The evaluation is bit-sliced and therefore the candidate is evaluated on the entire system term-by-term at once. The evaluation procedure can be found in `src/c/utils.c`.

If no solution was found, or an error occurred along the way, the procedure returns 1. Therefore, if a 1 is returned from a call to `solve()` the value in the `sol` pointer represents an invalid solution. Returning a zero conversely means success and that `sol` contains a valid solution.

5.3.2 FES

FES-recovery in C. Examining the `src/c/standard/fes.c` file many of the procedures have a corresponding implementation in the SageMath code. Procedures like `init()`, `update()`, `step()`, and the various bit-indexing functions are implemented by the same principles in both the C codebases and the SageMath codebase. Therefore, procedures like these will not be discussed greatly in this section.

Since the implementations of Algorithm 6 in C both solely make use of the `fes_recover()` approach from Chapter 4, this section will mainly describe the approach taken for the non-vectorized version of `fes_recover()`. Finally, at the end of the section a simple FES implementation for quadratic systems, as described in Section 2.4, will be described.

All FES-related procedures and declarations can be found in `inc/fes.h` and `src/c/standard/fes.c` (or `src/c/vectorized/fes.c`, for the vectorized version). Examining the declarations in the header file for non-vectorized compilation targets, one of the more important declarations is

(Listing 5.3.12) inc/fes.h

```c
typedef struct state
{
  poly_t i; /*! The value of the input variables assigned. */
  poly_t y;         /*! The evaluation of the system on s->i. */
  poly_t *d1;       /*! The first derivatives of the system evaluated on
  ↪   s->i. */
  poly_t *d2;       /*! The second derivative of the system (constants).
  ↪   */
  uint8_t *prefix; /*! The prefix used for the partial evaluation. */
} state;
```

being the state used by FES to store relevant values like derivatives and prefixes. Similarities to Listing 5.2.11 should be obvious. The `uint8_t *prefix` acts as an array in much the same way as the `prefix` attribute in Listing 5.2.11, although integers are limited to a single byte. This is more than sufficient, given that the solver could not feasibly solve systems with more than 256 variables, and so the positions saved in `uint8_t *prefix` would never reach such amounts.

Another important declaration is that of `fes_recover()`. A look into `inc/fes.h` would also show declarations for `fes()` and `bruteforce()`, however, as mentioned these procedures are explained near the end of this subsection. The `fes_recover()` declaration is quite similar to that of its SageMath counterpart

(Listing 5.3.13) inc/fes.h

```c
uint8_t fes_recover(poly_t *system, unsigned int n, unsigned int n1,
                    unsigned int deg, poly_t *results,
                    size_t *res_size);
```

The use of the `poly_t *` for storing solutions was already described and replaces the need for `res` array used in the SageMath implementation of `fes_recover()` (Listing 5.2.24). The return value is made to be an error code, whereas `poly_t *results` and `size_t *res_size` are out-parameters for the potential solutions found and the amount of these, respectively.

Inside `src/c/standard/fes.c`, the two top-most functions are `init_state()` and `destroy_state()`. These act like a constructor and a destructor, respectively, for the `state` struct shown in Listing 5.3.12. The constructor allocates heap memory according to the parameters of the problem instance, like n and n_1. Consequently, the memory has to be freed to avoid memory leaks, which is handled in `destroy_state()`. For anyone seeking to expand the codebase, the `init_state()` procedure will not take ownership over any passed pointers but will instead copy their data.

Next, in the same file, the `fes_recover()` definition is found. Although the similarities between the C implementation and Algorithm 9 (or the SageMath version in `src/sage/fes_rec.sage`) are noticeable, some areas still deserve an explanation. At the beginning of the procedure, the code initializes the different tables and structures needed

(Listing 5.3.14) src/c/standard/fes.c

```
458 state *s = NULL;
459
460 uint8_t *prefix = calloc((n - n1), sizeof(uint8_t));
461 if (!prefix) return 1;
462
463 size_t d_size = 0;
464 for (unsigned int i = 0; i <= deg; i++)
465 {
466   d_size += lk_binom[(n - n1) * BINOM_DIM2 + i];
467 }
468 poly_t *d = calloc(d_size, sizeof(poly_t)); //
469 if (!d) return 1;
470
471 unsigned int *alpha = calloc(deg, sizeof(unsigned int));
472 if (!alpha) return 1;
473
474 poly_t parities = INT_0;
```

The most interesting would be the derivative table `poly_t *d` allocated at Line 468. For the same reason as not storing potential solutions in a dictionary, the derivatives are not stored in one either. Instead, the table is a dynamically allocated array and the actual size of the array is stored in `size_t d_size`. The size is calculated as

$$d_{size} = \sum_{i=0}^{deg} \binom{n - n_1}{i}. \tag{5.2}$$

The indexing itself is computed using the `monomial_to_index()` function (from `src/c/standard/fes.c`), in order to work with the size of the table `d`:

(Listing 5.3.15) src/c/standard/fes.c

```
unsigned int monomial_to_index(poly_t mon, unsigned int n,
                                          unsigned int boundary)
{
  BEGIN_BENCH(g_index_time)
```

```c

  unsigned int i;
  int d = 0;
  unsigned int index = 0;
  unsigned int index_d = 0;
  for (i = 0; i <= boundary; i++)
    if (!INT_IS_ZERO(INT_IDX(mon, i)))
    {
      d++;

      index += lk_binom[i * BINOM_DIM2 + d];
      index_d += lk_binom[n * BINOM_DIM2 + d];
    }
  index = index_d - index;

  END_BENCH(g_index_time)

  return index;
```

The reasoning behind this different indexing scheme can be found in Section 5.4.1.

Once all tables and values have been set up, the procedure starts computing the *parities*, or evaluations, of the U polynomials. Like in the SageMath code, this is done using a procedure called `part_eval()`, handling `state` updates and calls to `fes_eval_parity()`. Now, notice that `fes_eval_parity()` its own separate procedure. Instead of combining this alternative FES version, where parities are computed instead of storing solutions, with `fes_eval_solutions()` the C code separates it, allowing for better readability and extendability. In the *standard* C codebase, the `fes_eval_solutions()` computes and stores solutions to the input system, while `fes_eval_parities()` computes the parities from Algorithm 8.

After parities have been computed and returned to the `fes_recover()` procedure, it directly checks the parities for a potential solution and does relevant translations if needed. Recall that the C codebase does not have a procedure like Algorithm 7, and therefore instead requires solution checking and *translation* somewhere else, being directly in `fes_-recover()` in this case. The code for checking solutions is

```c
(Listing 5.3.16) src/c/standard/fes.c
if (!INT_IS_ZERO(INT_IDX(parities, 0)))
{
  results[0] = GF2_ADD(0, INT_LSHIFT(INT_RSHIFT(GF2_ADD(parities,
  ↪   INT_MASK((n1 + 1))), 1), (n - n1)));
  (*res_size)++;
}
```

The above snippet shows how parities are handled for $\hat{\mathbf{y}} = \mathbf{0}$, whereas a similar snippet can be found for $\hat{\mathbf{y}}_i$, for $0 < i < 2^{n-n_1}$. Beware, the macros used in the conditional check are explained in Section 5.5. The if-statement checks for the case where U_0 evaluates to one (recall Section 3.3), as the parities and **y**-bits then form a potential solution to the system $\mathcal{P}$. If so, the the **y**-bits are stored as is, and the parities are converted into the corresponding **z**-bits. Then both sets of bits are combined into a single `poly_t`. The case for $\hat{\mathbf{y}}_i$, where $0 < i < 2^{n-n_1}$, replace `parities` for the `d[0]` entry in the if-statement and **z**-bits, as it now interpolates and evaluates using that directly. This of course also stores the counter value `si`, instead of 0, as the **y**-bits.

As already mentioned, much of the procedure works like Algorithm 9 and the SageMath counterpart. Therefore, the specifics of interpolation/evaluation are mostly the same. Computing the bit-positions required for interpreting the counter as a monomial (as Line 6 in Algorithm 9) is done through the `bits()` function, located in `src/c/standard/fes.c`. The positions are stored in an array `alpha` like the SageMath counterpart does it

(Listing 5.3.17) src/c/standard/fes.c

```c
unsigned int len_alpha = bits(si, alpha, deg);
```

which may then be used to compute indices using `monomial_to_index()` (Listing 5.3.15), as follows

(Listing 5.3.18) src/c/standard/fes.c

```c
unsigned int idx =
    (j == 0) ? 0 : monomial_to_index(si, n - n1, alpha[j - 1]);
```

In the interpolation phase, the **y**-bits are represented by an array of the bit-positions for each 1-bit in the counter-value (limited to the first $n - n_1$ bits). This is used for the `uint8_t *prefix` from Listing 5.3.12. Computing the prefix, the Gray code value of the counter `si` is used:

(Listing 5.3.19) src/c/standard/fes.c

```c
for (unsigned int pos = 0; pos < (n - n1); pos++)
{
  prefix[pos] = (1 & (gray_si >> pos));
}
```

Now, the rest of the interpolation phase is set up much like Listing 5.2.22, and will therefore not be explained in-depth. Due to the nuances of C, when compared to SageMath, there are certain less interesting differences. See Section 5.2.2 for an explanation of the SageMath code, Algorithm 9 for the general approach, and `src/c/standard/fes.c` for the standard C source code.

Should an error occur during execution of `fes_recover()`, the procedure cleans up its memory and returns 1, much like the error codes in the `solve()` procedure. In case no error occurred, the procedure returns 0.

FES for the curious. A simple implementation of the FES approach (for quadratic polynomials) discussed in Section 2.4 was also implemented. This exists mostly for comparison against Dinur's solver, however, it may still be used for solving systems. The remaining part of this subsection is dedicated to a brief look into this.

There are two methods for running FES directly in the C code; the `bruteforce()` procedure and `fes()` procedure. The `bruteforce()` procedure is an implementation of the `BRUTEFORCE()` procedure from Algorithm 8. The declaration can be found in `inc/fes.h` and the implementation in `src/c/standard/fes.c`. The procedure is declared as

(Listing 5.3.20) inc/fes.h

```c
unsigned int bruteforce(poly_t *system, unsigned int n, unsigned int n1,
                        unsigned int d, poly_t *solutions);
```

which is rather similar to that of Algorithm 8. Here, the `poly_t *solutions` parameter is an out-parameter for storing solutions. The return value is the number of solutions stored. This out-parameter of course expects the pointer to point to a *sufficiently*-sized chunk of memory for storing the solutions in.

If the goal is to run FES more in line with [4] or [3], the `fes()` function is defined to be able to handle this:

(Listing 5.3.21) inc/fes.h

```c
unsigned int fes(poly_t *system, unsigned int n, poly_t *solutions);
```

This procedure is not as heavily optimized as the GPU version of [3], or the FPGA version of [4], but has received the same level of *care* as the procedure `fes_eval_parity()` used inside `fes_recover()`. The solutions found are stored in the parameter `poly_t *solutions` and the return value holds the amount found.

5.3.3 Utilities and benchmarking

Besides the C procedure discussed up to this point, a few more can be found in `src/c/standard/utils.c` and `src/c/standard/benchmark.c`. The most notable utility procedures are the `gen_matrix()` procedure and the `eval()` procedure. The former was briefly described in Section 5.3.1, whereas the latter is simply a procedure that evaluates a bit-sliced system on some *full* assignment of variables, in parallel. Declarations for these procedures can be found in `inc/utils.h`, if needed for extending or reusing the codebase.

When the shared library is compiled, a few benchmarking tools are also available. First off, if the goal is to solely benchmark the solver, the procedure

```
(Listing 5.3.22) inc/benchmark.h
void e2e_benchmark(size_t rounds, poly_t *systems[], size_t n, size_t m);
```

takes as input a list of bit-sliced systems that will then be run through the `solve()` function. The procedure computes timings for different points of interest in the codebase but also computes the number of potential solutions stored, as well as the balance of interpolation versus evaluation in `fes_recover()`. All of the computed values are averaged according to how many calls to `solve()` that finished successfully, i.e. returned a valid solution. The stored timings and iteration counts can be accessed through the global variables declared in `inc/benchmark.h`, prefixed with a `g_`.

If a non-vectorized variant of the codebase was compiled, a benchmarking procedure for the `fes()` procedure is also available. The declaration is as

```
(Listing 5.3.23) inc/benchmark.h
void fes_benchmark(size_t rounds, poly_t *systems[], size_t n, size_t m);
```

This procedure acts much the same way as `e2e_benchmark()`, just now solely for FES. Computed timings are stored in separate global variables than those used by `e2e_benchmark()`.

Benchmark timings are computed using the `BEGIN_BENCH()` and `END_BENCH()` macros, as well as `READ_BENCH()`. The names should make the purposes of the macros more understandable. Should there be interest in adding more benchmarks; declaring an extern variable in `inc/benchmark.h` for storing timings (or another type of benchmark value), initializing it to a suitable value in a user-chosen position, and then adding appropriate macro-calls at the points of interest is sufficient.

5.4 Optimizations

Different kinds of optimizations were implemented either directly in the codebase described in Section 5.3 or in the alternative codebase in `src/c/vectorized/`. This subsection seeks to go through the most prominent optimizations and why they are used.

5.4.1 FES-based recovery

Internal optimizations to FES-recover. For the C codebases, the choice of procedures for interpolating and evaluating the U polynomials was `fes_recover()` and its vectorized counterpart. One benefit of going with `fes_recover()` and `fes_recover_vectorized()` is that interpolation and evaluation of these polynomials can be done in *one* pass-through of their 2^{n-n_1} evaluations, instead of two separate calls to the Möbius transform.

Another effect of using the `fes_recover()` (this time, non-vectorized) approach was the ability to more easily store only the potential solutions, instead of allocating memory for all evaluations. Alternatively, as is described in Section 3.3, the procedure would have to store all

$$2^{n-n_1}$$

evaluations. In [9] it was shown that the number of suggested solutions in a round was about

$$2^{n-2n_1},$$

and so the saved memory could prove a benefit for larger values of n_1. Although the current implementation does allocate a large chunk of memory at once, memory is reallocated once the amount of potential solutions is known. Other alternatives for benefitting from compactly storing potential solutions also exist, depending on what the end goal is.

Another memory-saving element used in `fes_recover()` is the alternative indexing scheme used in the C code, compared to that of the SageMath code. This alternative indexing was described in Section 5.3.2. Reusing the SageMath code, the derivative table `d` would have to be a large exponential in size, as monomial representations would be sparsely scattered depending on their bit-string representation. The approach taken instead yields a memory consumption of

$$\sum_{i=0}^{d_{degree}} \binom{n-n_1}{i} < \left(\frac{(n-n_1)e}{d_{degree}}\right)^{d_{degree}}$$

where d_{degree} is the degree passed to `fes_recover()` as `deg`. Now, this is still an exponential function in n, as d_{degree} (or `deg`) is a function of n. However, this exponential function has a much smaller growth in theory. For actual data on memory usage, see Section 6.1. Of course, the disadvantage of this approach is the additional computational overhead used to compute the different indices, compared to the more *naive* approach chosen in the SageMath prototype (Section 5.2.2).

In total, the memory consumption of `fes_recover()` is lowered, and in expectation, it is rather competitive to using in-place Möbius transforms like Fig. 2.5. Using the memory-efficient Möbius transform suggested in [9], could pose a strong alternative to this approach.

Removing `output_potentials` Although the SageMath prototype kept `output_potentials()`, the choice was to leave it out in the C code. Due to the choice of using the evaluation and interpolation process of Section 4.1, integrating solution filtering and translation directly into the procedure was more intuitive.

Since each iteration of the outer-most loop of Algorithm 9 (and thereby its implementations), discretely, either evaluate or interpolate, integrating filtering and translation directly into `fes_recover()` was quite simple. Other than allowing for tighter packing of potential solutions in the returned array, this process also removes the need for a separate loop that filters and translates by going through all 2^{n-n_1} evaluations (Line 11 in Algorithm 7).

Therefore, the time taken to go through all solutions once more, in order to *post-process*, is no longer effecting the total solve time. Of course, as already mentioned, this comes at the cost of potentially having a heavier computational load at each iteration of `fes_recover()`.

5.4.2 Vectorization and parallelization

Various methods of parallelizing the procedure were used, including the use of SIMD instructions and multicore CPUs. Of course, the shared library is compiled with GCCs `-O3` flag, which implies quite a few optimizations occur at compile-time as well. However, some optimizations can be hard to determine statically and so active care was taken to provide further optimizations.

A smaller method of optimizing storage is the use of the different Make flags available. The Make flags can currently tell the solver what size of integer the input systems (bit-sliced) and their solutions should be stored in. This can help reduce the overall memory footprint. Currently, the vectorized implementation divides $\mathcal{P}$ and $\tilde{\mathcal{P}}_k$ into separate integer sizes, which is a feature that is not readily available in the non-vectorized version. Using smaller integer sizes for the smaller systems $\tilde{\mathcal{P}}_k$ allows for storing more polynomial terms in a cacheline, boosting performance to some extent. As these terms are accessed quite frequently, and in an almost sequential manner, this can boost performance on a low level. Therefore, having separate integer sizes for $\mathcal{P}$, $\tilde{\mathcal{P}}_k$, and possibly also solutions, could prove beneficial for the standard version as well.

Although it is possible to vary integer sizes for both the standard and vectorized implementations, both are limited to input systems of 64 polynomials and 64 variables (maximally). This may be extended, see Section 5.5.

Bit-slicing. As bit-slicing is a simple method of obtaining linear speedups, especially when used in the context of systems of boolean polynomials, it was a safe choice in this case. Since Dinur's polynomial-method algorithm can work with multiple polynomials at once in almost every area, many of the procedures and sub-procedures benefit from this.

In both the non-vectorized builds and the vectorized builds, the input system to `solve()` is bit-sliced (mentioned in Section 5.3.1). The sub-system $\tilde{\mathcal{P}}_k$ is generated in `compute_-p_k()` (both the version in `src/c/standard/mq.c` and `src/c/vectorized/mq.c`) bit-sliced as well. The multiplication of individual matrix indices, recall Eq. (5.1), with terms of the polynomials of $\mathcal{P}$ is quite easily implemented with bit-wise operations. Using bit-slicing and bit-wise `and` is then simply a parallel multiplication. Then the summation is a matter of computing the parity of the integer representing said multiplication.

Also in `fes_recover()` and `fes_recover_vectorized()`, bit-slicing benefits the internal calls to `fes_eval_parity()` as well, through the evaluations of the n_1+1 polynomials; U. The evaluations of these polynomials, i.e. the parities computed in `fes_eval_-parity()` and in the evaluation phase of Algorithm 9, are stored and computed bit-sliced as well. Processing and checking for candidates is then merely a matter of bit-masking.

Generally, most steps of Dinur's polynomial-method solver benefit from bit-slicing, as many operations can be done independently in parallel. The polynomials and systems are not

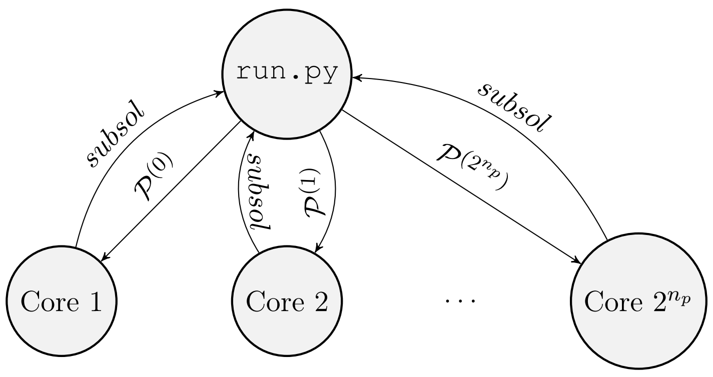

Figure 5.1: Visual example of the parallelization strategy for solving a system $\mathcal{P}$ on multicore architectures. Each $\mathcal{P}^{(i)}$ is a partially evaluated system. Once a core returns a solution, the `run.py` script appends to it the values of the fixture for that core. Here, $n_p = \lfloor \log_2 q \rfloor$ and q is the amount of cores.

always the same, as it was just explained that both operations on $\mathcal{P}$ and the U polynomials can benefit from this.

Multicore. Another optimization strategy is the use of multicore CPUs. In order to solve larger systems, a good solver would make use of as many of the resources available as possible. Through the `run.py` script, the solver can be spawned in a multicore context, wherein each core runs a separate *instance* of the input system.

Say the CPU(s) of the system has (have) a combined q cores available. By fixing

$$n_p = \lfloor \log_2 q \rfloor$$

variables, 2^{n_p} solvers can be run in parallel by *partially evaluating* the input system $\mathcal{P}$. Recall from Section 2.2.1 that with n_p variables, 2^{n_p} variable assignments are possible. Therefore, by fixing n_p variables, 2^{n_p} cores may receive their own partially evaluated system, i.e. an independent input system. That is, a partial evaluation would generate systems,

$$\mathcal{P}^{(0)}, \mathcal{P}^{(1)}, \ldots \mathcal{P}^{(2^{n_p})},$$

that can be solved independently. This idea is also described in Section 2.2.1. If one of the 2^{n_p} cores find a solution, the core can simply append the fixture of the n_p variables to obtain a solution for $\mathcal{P}$.

Currently, the multicore operation is implemented directly in the `run.py` script, using Pythons built-in `multiprocess` library. Using this library for concurrency ensures that no problems with Python's infamous Global Interpreter Lock are met. However, the variables for the input system are fixed in Python/SageMath, handing each fixed system to a separate process that then internally calls the `solve()` function from the shared library (either vectorized or standard, depending on what was compiled).

A visualization of the multicore implementation can be found in Fig. 5.1.

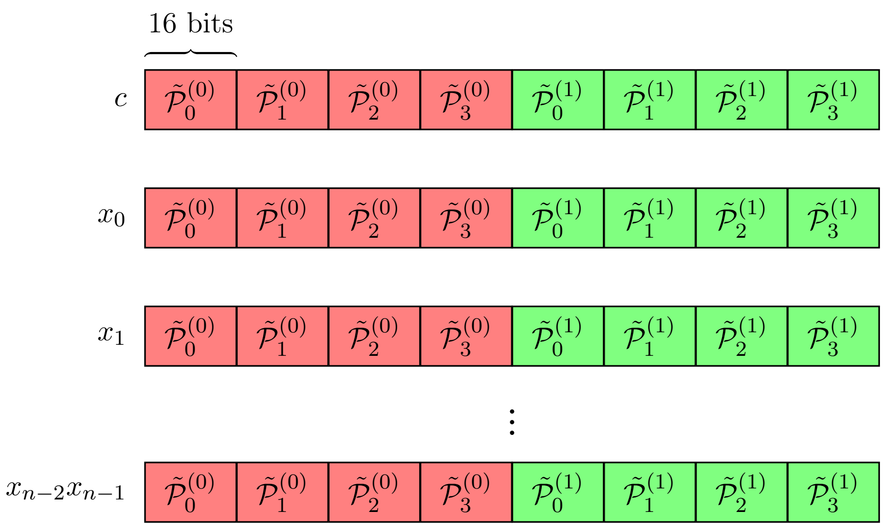
16 bits
c
x_0
x_1
$x_{n-2}x_{n-1}$
$\tilde{\mathcal{P}}_0^{(0)}$ $\tilde{\mathcal{P}}_1^{(0)}$ $\tilde{\mathcal{P}}_2^{(0)}$ $\tilde{\mathcal{P}}_3^{(0)}$ $\tilde{\mathcal{P}}_0^{(1)}$ $\tilde{\mathcal{P}}_1^{(1)}$ $\tilde{\mathcal{P}}_2^{(1)}$ $\tilde{\mathcal{P}}_3^{(1)}$

Figure 5.2: Visualization of how the $\tilde{\mathcal{P}}_k^{(i)}$ sub-systems are parallelized in a 128-bit AVX register, using integer sizes of 16-bits ($n_1 + 1 \leq 16$). This case would fix 1 variable, creating two new systems $\mathcal{P}^{(0)}$ and $\mathcal{P}^{(1)}$, each with their sub-systems, $\tilde{\mathcal{P}}_k^{(i)}$. One row above corresponds to how `poly_vec_t` types are used in the vectorized codebase. The structure for the derivative table entries is similar to a row above.

SIMD instructions. As the target machines for this solver are x86-based, many SIMD instructions are available. In [9], the memory-efficient Möbius transform is described alongside a theoretical optimization strategy. The idea for this strategy was to pre-compute multiple $\tilde{\mathcal{P}}_k$ systems and interleave their respective evaluations of the U polynomials with solution testing. This idea is somewhat mimicked with the use of SIMD instructions and `fes_recover_vectorized()`.

Using appropriate Make-flags (see Section 5.7) the `bin/mq.so` library may either include AVX or AVX2-based instructions (specifically as GCC intrinsics). Now, the vector registers available in modern x86 CPUs, are either 128 bits or 256 bits in size. These registers may contain different sizes of integer or floating-point data internally. For this solver, integer data is of the most relevance.

Since the `solve()` function internally sets

$$n_1 = \left\lceil \frac{n}{5.4} \right\rceil,$$

and the $\tilde{\mathcal{P}}_k$ systems consist of $\ell = n_1 + 1$ polynomials, an input system of $m \leq 37$ can fit bit-sliced terms of $\tilde{\mathcal{P}}_k$ into single bytes. For these system sizes it is possible to pack either $\frac{128}{8} = 16$, or $\frac{256}{8} = 32$, bit-sliced terms of $\tilde{\mathcal{P}}_k$s into a single AVX register. Likewise, for $\mathcal{P}$ with $37 < m \leq 64$, an AVX register can hold either $\frac{128}{16} = 8$ or $\frac{256}{16} = 16$ terms of $\tilde{\mathcal{P}}_k$s in separate 16-bit integers inside the vector register.

Given that the expected number of rounds to find a solution is 4 (shown in [9]), the vectorized version of the `solve()` function uses the AVX registers as follows: Say an AVX register can hold terms for j sub-systems $\tilde{\mathcal{P}}_k$. By fixing

$$n_p = \log_2 \frac{j}{4}$$

variables of $\mathcal{P}$, each partially evaluated version of $\mathcal{P}$ gets its own four vector indices, for the $\tilde{\mathcal{P}}_k$ systems. That is, the procedure fixes n_p variables, producing the systems

$$\mathcal{P}^{(0)}, \mathcal{P}^{(1)}, \dots \mathcal{P}^{(2^{n_p})}.$$

Then, each round of `solve()` computes four $\tilde{\mathcal{P}}_k^{(i)}$ systems for each fixed system $\mathcal{P}^{(i)}$ and stores them all in one AVX register. This way, AVX registers can hold a certain term for each bit-sliced system internally as integer data. This allows the procedure to operate in a lock-step manner on multiple systems at once, without having to synchronize between threads or processes. A visualization of this parallelization strategy can be seen in Fig. 5.2.

For the most part, the procedure follows the likes of the *standard* implementation described in Section 5.3, however, some parts may need explanation. In the setup of `solve()` in `src/c/vectorized/mq_vectorized.c`, the procedure fixes the input system $\mathcal{P}$ and stores the fixed systems in new and smaller integer arrays

```
(Listing 5.4.1) src/c/vectorized/mq_vectorized.c
poly_t assignment[FIXED_VARS] = {0};
for (poly_t i = 0; i < (1 << FIXED_VARS); i++)
{
  for (int j = 0; j < FIXED_VARS; j++) assignment[j] = (i >> j) & 1;
  fix_poly(system, fixed_system + i * sys_len, assignment, new_n, n);
}
```

which can be seen in the snippet above. The macro `FIXED_VARS` is set depending on the Make flags (see Section 5.7), but represents the number of variables fixed for the current setup AVX setup. The variable `new_n` is set to be

```
(Listing 5.4.2) src/c/vectorized/mq_vectorized.c
unsigned int new_n = n - FIXED_VARS;
```

which then is used internally instead of the original `n` variable for most computations. Of course, `FIXED_VARS` represents the value of n_p from earlier, chosen through the flags given to Make at compile time.

The procedure `gen_matrix()` is kept the same (`src/c/utils.c`), primarily to comply with the testing framework. However, instead of computing only one matrix in each round of `solve()`, the procedure computes one for each $\tilde{\mathcal{P}}_k^{(i)}$ in the vector register. The `rand_sys` variable, traditionally representing $\tilde{\mathcal{P}}_k$, is however a bit different. Instead of `poly_t *rand_sys`, the generated systems are stored as `poly_vec_t *`:

```
(Listing 5.4.3) src/c/vectorized/mq_vectorized.c
poly_vec_t *rand_sys = aligned_alloc(ALIGNMENT, sys_len *
↪  sizeof(poly_vec_t));
```

A `poly_vec_t` is just a typedef for the AVX vector types in C, like `__mm128i`. The definitions can be found in `inc/mq_config.h`. Also noticed the `aligned_alloc` call. An effect of using AVX registers is that in any memory location storing vector values, the memory needs to be correctly aligned. Instructions do exist to work with unaligned memory, however, these are slower than their aligned counterparts.

The `fes_recover_vectorized()` is also different for the vectorized versions. The function declaration can still be found in `inc/fes.h`, but it looks a bit different:

```
(Listing 5.4.4) inc/fes.h
uint8_t fes_recover_vectorized(poly_t *system, poly_vec_t *e_k_systems,
                               unsigned int n, unsigned int n1,
                               ↪  poly_vec_t deg,
                               poly_t *result);
```

The procedure takes both the original system `poly_t *system` (non-fixed) as well as the array of vectors containing $\tilde{\mathcal{P}}_k^{(i)}$ systems. Also, instead of an array in `poly_t *result`, the procedure uses the parameter to return a single *actual* solution to $\mathcal{P}$.

Going into `src/c/vectorized/fes.c`, much of the code may seem similar to that explained in Section 5.3. Much of the logic is on an abstract level the same, but instead of using `poly_t *` to represent systems, most procedures instead use `poly_vec_t *`. The various function-like macros prepended with `VEC_` represent an abstraction layer over the AVX intrinsics used to work with AVX registers. Recall that each AVX register contains four rounds worth of sub-systems, for evaluations of $\mathcal{P}$ on some fixture. Therefore, snippets like

```
(Listing 5.4.5) src/c/vectorized/fes_vectorized.c
sub_poly_t z = GRAY(s->i);

zero_mask = VEC_IS_ZERO(s->y);

added = VEC_SETBIT(*parities, 0, 1);

*parities = VEC_BLEND(*parities, added, zero_mask);
```

display how the aforementioned ”lock-stepping” works in practice. The non-vectorized version of the above snippet is

```
(Listing 5.4.6) src/c/standard/fes.c
poly_t z = GF2_ADD(s->i, INT_RSHIFT(s->i, 1));

if (INT_IS_ZERO(s->y))
{
  *parities = INT_SETBIT(*parities, 0, 1);
```

for which the differences between the vectorized and non-vectorized versions show. Both snippets are from `fes_eval_parity()` in the vectorized and non-vectorized versions, respectively. Since the vectorized version runs multiple bit-sliced systems in parallel during the execution, conditionals have to be handled differently. E.g. for each system in the vector `s->y`, in Listing 5.4.5, a combination of AVX *masks* and *blends* are used to mimic multiple parallel conditional checks at once. These principles are the same throughout `src/c/vectorized/fes_vectorized.c`.

To quickly summarize the idea of AVX masks and blends: An AVX mask represents the evaluation of some logical statement in each of the vector elements, e.g. comparing equality between two vector registers would return a mask where the indices with *equality* store all 1-bits and those with *inequality* store all 0-bits. A use-case for such an AVX mask is *blending*, in which elements between two vectors are blended depending on some specified mask. This blending can also be interpreted as a *filtering* mechanism with a vector of ”defaults”.

At last, the biggest difference between the two codebases is how solutions, checking and processing, are handled. The snippet

```
(Listing 5.4.7) src/c/vectorized/fes_vectorized.c
if (_avx_sol_overlap(parities))
{
  poly_t solution, fixed_solution;
  PotentialSolution potential_solutions[(1 << FIXED_VARS) * 6] = {0};

  int amount = _avx_extract_sol(parities, potential_solutions);

  for (int count = 0; count < amount; count++)
  {
    fixed_solution = GF2_ADD(
        0,
        INT_LSHIFT(GF2_ADD(potential_solutions[count].solution,
        ↪  INT_MASK(n1)),
                   (n - n1)));

    solution = GF2_ADD(potential_solutions[count].fixed_var,
                       INT_LSHIFT(fixed_solution, FIXED_VARS));

    if (!eval(system, n + FIXED_VARS, solution))
    {
      *result = solution;

      destroy_state(s);
      free(prefix);
      free(alpha);
      free(d);

      return 0;
    }
  }
}
```

shows how the vectorized codebase handles evaluations of U polynomials. As an AVX register contains multiple $\tilde{\mathcal{P}}_k^{(i)}$ systems it computes the evaluations of the U polynomials in lock-step as well. Therefore, for each evaluation of the U polynomials, the system may simply check if overlapping values exist in each group of four vector elements in the register. If an overlap is found, the procedure extracts all overlapping solutions directly from the register, constructs the solution(s) and evaluates on $\mathcal{P}$ directly. If a solution is found, the procedure may return. If no full evaluation on $\mathcal{P}$ succeeds, the procedure returns an error, which tells `solve()` to start a new round.

Because evaluations of U the polynomials are computed and checked simultaneously, no large lists of solutions are stored. This will drastically affect memory consumption and entirely mitigates the need for storing solutions. The cost of this is that the procedure potentially skips some solutions as each round of `solve()` in `src/c/vectorized/mq_-vectorized.c` logically corresponds to computing four rounds in the *standard* version and throwing away all potential solutions found if none was found in said group of four rounds.

5.5 C abstractions

The C codebases contain certain levels of abstractions, in order to make them more extendable. One example of this is the use of macros. Recall from previous subsections that many places in the C code, macro-like functions such as `INT_IS_ZERO`, `INT_IDX`, or `GF2_MUL` are called (Listing 5.3.15, Listing 5.3.11 to give some example listings). These macros are used as an abstractional layer in order to hide the structure of the underlying `poly_t` type. If the goal is to extend the codebase to support wider integer types, allowing for larger systems bit-sliced into integers, these macros help hide the operands used. This could be used in case the wider integer type does not support certain operations. E.g., GCC has support for 128-bit integers on target machines with wide enough integer modes, however, these do have limitations in certain areas and may therefore require workarounds that can be hidden using the aforementioned macros. The macros defined for integers can be found in `inc/mq_uni.h`.

Likewise, the vectorized codebase hides many operations behind macros for the same reason. These macros can be found in `inc/mq_vec.h` which internally makes use of either `inc/vec128_config.h` or `inc/vec256_config.h`. The latter two header files also include relevant macros for stating how many variables to fix given the Make flags used, the number of elements in a vector register (given the integer sizes stored in it), and other constants. Say the goal is to extend the codebase to use AVX512 as well, or future AVX (maybe even Intels AMX instructions): The process is a matter of creating a file similar to `inc/vec128_config.h` or `inc/vec256_config.h`, adding an appropriate conditional case in `inc/mq_vec.h`, and finally running the code. Exactly which macros need to be present in the newly added vector-config header is described in `inc/mq_vec.h`.

Depending on how exotic the vector instruction set is, it may be necessary to add or change functionality in `src/c/vectorized/vector_utils.h` as well. If it is simply a matter of adding support for a newer version of AVX, then the approach just described should be fine.

Ensuring that macros like `FIXED_VARS` and `VECTOR_ELEMENTS` (from `inc/vec128_-config.h` and `inc/vec256_config.h`)are set correctly, ensures that the vectorized codebase is automatically set up, at compile time, for fixing the correct amount of variables and only storing four rounds for each fixed polynomial, as described in Section 5.4.

5.6 Testing the code

The following subsection contains some brief notes on the testing methodology and how it was executed. For more on how to run and test different areas of the codebase, see the `README.md` file in the accompanying Git repository.

5.6.1 SageMath

For each of the procedures implemented in Python or SageMath, the accompanying test procedures can be found in the same file as the procedure being tested. For example, the tests related to `bruteforce()` and `fes_eval()` can be found in `src/sage/fes.sage`, while tests related to `solve()`, `output_potentials()` and `compute_u_values()` are found in `src/sage/dinur.sage`.

Since this SageMath prototype would eventually act as a reference point for the C implementation, the testing approach was to essentially create unit tests for select parts of the codebase. For procedures such as `output_potentials()`, the testing methodology was to evaluate the U polynomials in their entirety. The following snippet shows an example,

```
(Listing 5.6.1) src/sage/dinur.sage
si = sum(F_tilde(*convert(y, n - n1), *convert(z_hat, n1 - 1)[:i - 1], 0,
↪  *convert(z_hat, n1 - 1)[i - 1:]) for z_hat in range(2^(n1 - 1)))
```

The snippet above is directly computing the sums related to the U_i polynomials ($i = 1, \dots n_1$). This testing strategy, where values for the theoretical constructs are explicitly computed and compared against the outputs of their target procedures, is repeated whenever possible. This way, the testing process is not bound to only run work with pre-computed results. In addition, the testing framework allows for generating multiple systems and storing any input system that triggers a fail in a test, so it may be reused later. A downside to this approach is of course that some of these theoretical constructs take quite a while to compute.

These tests can be executed using the `run.py` script with the `-t` flag, specifying what test should be run. To list all available tests, use the `-l` flag. Information about the `run.py` script can be found in the accompanying `README.md` file and partly in Section 5.8.

5.6.2 C code

For the C implementation, there are different levels of tests. Either one may compile the `bin/test` binary, using the appropriate Make flags and targets (see Section 5.7), or one may run tests using the shared library `bin/mq.so` (if compiled). In both cases, the results of the C implementations are compared against relevant reference points in the SageMath code implementations. This way, going from the unit tests for the SageMath code, the C code can be directly compared to the verified SageMath code. The test functions for the C code may also be run via the `run.py` script.

Running tests with the `bin/test` executable file, memory sanitizers and debugging information is enabled. These tests are used for testing procedures like the computation of the $\tilde{\mathcal{P}}_k$ sub-systems and comparing it against the SageMath version. In general, these tests are fed relevant inputs via the SageMath code as well as the desired result(s) and checks against them internally in the executable. All these tests are found in `src/sage/dinur.sage` and contain a postfix of `_SAN` in their function name.

The alternative is to test different functions using the `bin/mq.so` shared library. Other than testing for correctness, these tests also help test the bridge between the shared library and Python/SageMath. Like the tests for the SageMath code itself, these tests can be found in the related `.sage` files; tests for `fes()` and `bruteforce()` can be found in `fes.sage`, `fes_recover()` in `src/sage/fes_rec.sage`, etc. The tests related to the C codebase(s) all contain the character c in test names.

It could be argued here that more tests should exist, especially for the C codebase. Many new procedures were introduced in both the *standard* and *vectorized* codebases, implying that further testing of these individual parts should have taken place on a more granular level.

5.7 Compilation and compile-time parameters

The accompanying Makefile can conform to multiple platforms using either vectorized instructions or building for machines with different register sizes. Building any of these different targets requires altering the `REGSIZE` flag, and/or the `INTSIZE` flag when calling `make`. Setting `REGSIZE` to 8, 16, 32, or 64 means building the non-vectorized version, but regulates the integer sizes used to store the polynomials and solutions to the given width. Specifying 128 or 256 means that the build uses 128-bit or 256-bit registers, respectively. When specifying `REGSIZE` as one of the vector sizes, the `INTSIZE` flag controls the size of integers stored in these AVX vectors. The options for `INTSIZE` on vectorized builds is either `INTSIZE=8` or `INTSIZE=16`.

The default target creates the file `bin/mq.so`, ready for dynamic linking into other projects, optimized with `-O3`. The contents of this shared library are described in Section 5.3. Running `make tests` will create an executable `bin/test` with memory sanitizers and debug flags enabled.

While compiling any of the targets, the makefile generates a few extra files as well. One file is `src/sage/.compile_config`, ensuring better interoperability between the SageMath code and the C code. A more prominent file is the `inc/binom.h` file generated. The `inc/binom.h` file is a header file containing a lookup table *of sufficient size* alongside the necessary macros to do lookups quickly. This lookup table is generated by the Makefile which internally calls the `binom.py` script and saves the output in `inc/binom.h`. The lookup table is used by the indexing function `monomial_to_index()`.

5.8 Interface for running the C code

The C code was developed to be usable via multiple means. Once a compiled shared library is available, the code can either be loaded into other projects that support `.so` files, or it can be run as a complete solution via `run.py`.

Running the solver as a script, an invocation without any flags results in a single-core solve routine using the version of the build that was specified at compile time (in essence; *standard* or *vectorized*). The invocation will either ask for a path to an MQ-challenge[2] style text file from which the system may be loaded, or for relevant parameters in order to generate the systems before solving. Giving the `-p` or `-parallelize` flag when invoking `run.py` allows the script to parallelize the solver by fixing variables according to how many CPU cores are present on the system. This last part was described in Section 5.4.

Using the shared library on its own in an independent project is also quite straightforward. Although quite a few header files are present in the `inc/` folder, the most important one is `inc/mq.h`. If instead the goal is to use other parts of the codebase one may include `inc/fes.h` for FES-related declarations, or `inc/utils.h` for different utilities like evaluating polynomials, generating matrices or simply computing indices for the bit-sliced

[2] `https://www.mqchallenge.org/`

polynomials. The `inc/vector_utils.h` header is for utilities for the vectorized implementations. Definitions and macros used throughout the codebase may be included via the `inc/mq_config.h` file. At last, the benchmarking code can also be called on the shared library file. This provides the ability to hook in and read benchmarks directly in C code (from the global variables), and of course, call the benchmark procedure as a whole. Relevant benchmark declarations are stored in `inc/benchmark.h`. As the benchmark values are simply stored as global variables, it is not recommended to do much more than read the variables.

Note, common for any function that takes a `poly_t` array (or `poly_vec_t`) as input is that the stored system is expected to be bit-sliced, linearized, and (monomials) stored in *graded lexicographic order*.

The C code can technically also be executed via the `bin/test` executable, which may be compiled with the accompanying Makefile. However, this executable is made to primarily work in tandem with tests in the SageMath code, therefore it is probably not of great interest to execute the solver this way.

Chapter 6

Evaluation

As has been acknowledged multiple times, memory consumption is one of the largest bottlenecks for Dinur's polynomial-method solver. From [9], it is known that the procedure will store 2^{n-2n_1} solutions in each round, in expectation. Workarounds for this were also introduced in that same paper, with alternatives introduced here as well. Therefore, Section 6.1 is dedicated to evaluating the memory consumption of the current implementation. Further, since the effectiveness of the procedure was shown to be largest for rather large polynomial systems, Section 6.2 looks at how the implementation fares against an alternative procedure that has been tested and used in practice more than a few times. At last, Section 6.3 examines how the implementation handles "larger" parameter sizes.

6.1 Memory consumption

For a visual comparison of the three most prominent C variants, refer to Fig. 6.1. The figure displays the memory consumption on these variants, in `MiB`, across the time they spent solving the same system. The system can be found in the `report/metrics/memory/` folder, alongside the `.dat` files used for plotting this figure and the `procinfo_` files used later in this subsection.

It should be noted that the graphs in Fig. 6.1 are all measured via the `run.py` python script, meaning that some amount of memory will also be allocated for Python and SageMath related elements, like the interpreter. However, large parts of the SageMath-induced memory consumption also stem from things like storing the input system as a list of SageMath polynomials prior to calling the shared library. This would potentially be better if a frontend was developed in a language more similar to C. From testing, around 200-230 `MiB` are used up until the C library is called when running it on systems like those used in Fig. 6.1.

6.1.1 Standard

Inspecting graph (a) in Fig. 6.1, it is quite clear that the solver spent three rounds to solve the entire system. The memory consumption gradually increases up until around 30 seconds,

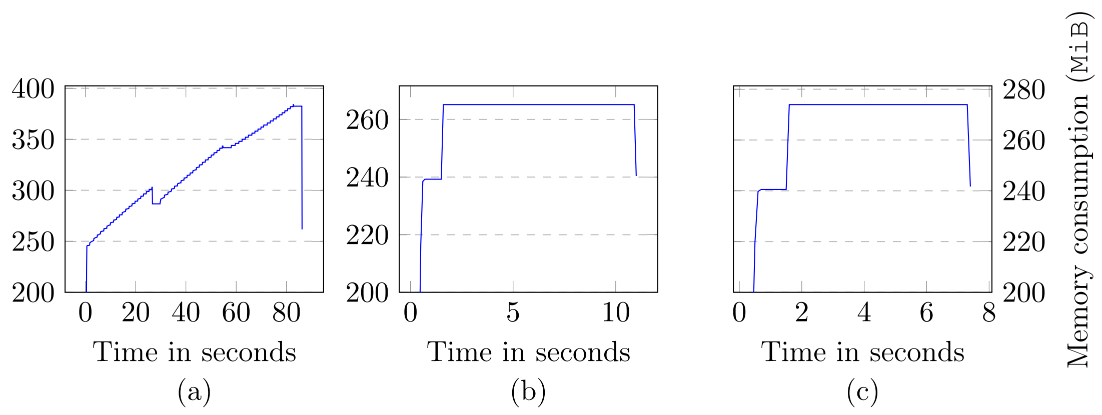

Figure 6.1: Memory consumption for three variants of the C implementation on a system with $n = m = 31$. (a) *Standard* C version; compiled to use 32-bit registers. (b) SIMD version; 128-bit AVX registers. (c) SIMD version; 256-bit AVX registers.

at which point it falls and not long after starts increasing again. Similar ”hitches” in the increasing memory consumption can be seen at around 50-60 seconds. These increases are as expected, given that the procedure has to store large amounts of solutions at each round, comparing newly found potentials to previous ones.

The expected amount of memory for storing solutions is about $4(n_1 + 1) \cdot 2^{n-n_1}$ bits, in expectation (theoretically). Of course, this comes stems from allocating full $(n_1+1)\times 2^{n-n_1}$-sized tables for storing candidate solutions in each round. Storing these bits in individual bytes would yield a memory consumption of

$$\frac{4 \cdot 8 \cdot 2^{31-7}}{2^{20}} = 512$$

`MiB` in total, for a naive implementation (for the system size used in Fig. 6.1). In comparison, storing the solutions bit-sliced, combining the $n_1 + 1$ bits into a single byte would yield a 64 `MiB` memory consumption. At last, storing only candidate solutions would yield

$$\frac{2^{31-2\cdot 7} \cdot 4}{2^{20}} = 0.5$$

`MiB` of memory, in expectation. This last calculation assumes that all 31 bits of a solution are stored in a single 32-bit integer. Now, remember, these calculations are purely theoretical as they do not take into account elements like memory alignment, storage of `structs`, stack usage, and memory consumption as a whole.

Since candidate solutions are stored using an array of `poly_t`, the memory allocated for storing solutions is rather close to 4 bytes per stored solution. In `report/metric-s/memory/bench_output` other data from the standard run can be found. From this file, the number of stored solutions can be read, which totaled 31 952 544 solutions. This means that around

$$\frac{31\,952\,544 \cdot 4}{2^{20}} \approx 122$$

`MiBs` were used for storing solutions alone. This amount of storage fits quite nicely with what can be derived from Fig. 6.1a, but is not comparable to the $2^{31-2\cdot 7}$ expected potentials. Running the standard solver on similar system sizes yields the same results. Due to time constraints, the exact condition triggering this was not discovered. Some plausible factors could be elements like sampling of system polynomials, sampling of random matrices or various analytical assumptions made when proving the complexity of the algorithm that might not have been fulfilled. This latter case can relate to the two former ones, as the analysis in [9] made certain assumptions that may not be entirely met in practice (on the distribution of system samples).

Nevertheless, from the theoretical 512 `MiB` of storage, the standard implementation manages to cut down the memory consumption, based on the experimental evidence. Also note, even though bit-slicing the $(n_1+1)\times 2^{n-n_1}$ table theoretically yields a smaller memory consumption than the experimental one, for the same parameter sizes, this does prove a time-memory trade-off. Packing solutions sequentially can yield more cache-friendliness, especially considering that the access patterns in the table of solutions are quite linear no matter how the solutions are iterated. Another point is that a $(n_1 + 1) \times 2^{n-n_1}$ (even if bit-sliced) requires one to potentially iterate all 2^{n-n_1} keys, instead of only ever accessing the relevant solutions.

In Fig. 6.2, a more general comparison of memory strategies is shown. The curves are shown up to $n = 32$, since the samples stem from an implementation using 32-bit integers for solutions. The *full-sized* and *bit-sliced* curves represent the theoretical values as a function of n, where $n_1 = \lceil \frac{n}{5.4} \rceil$. The third curve is fitted from averages across five runs for 10 different parameter sets. Clearly, this more general picture shows the same story as the single sample from Fig. 6.1. Of course, the curve may be more accurate if more samples were used, or the averages computed for more systems. Therefore, Fig. 6.2 should not be taken as an exact science, but does hint at the aforementioned trend in the relationship between the three memory strategies. Also, the major *hitches* in the blue and green curves stem directly from the value of n_1 changing, which would yield quite different results if the $\frac{n}{2.7d}$ were used specifically (as proposed in [9]).

Now, to sum up, the memory consumption of the standard implementation: The implementation acts much as expected theoretically, given how solutions are stored. Although the consumed memory is less than a by-the-books storage mechanism, the trend is still exponential and therefore a great bottleneck. With *compact* storage mechanism used for solutions, it uses more memory than if elements were bit-sliced. If $n > 32$, the extrapolated red curve would need to be re-fitted on samples from an implementation using 64-bit integers, which would also drastically increase memory consumption (potentially 31 wasted bits in each integer). The two theoretical curves would also need to be revised when $n > 37$, as the evaluations of the U polynomials ($n_1 + 1$ bits) no longer fits into a byte. However, the general nature is still tending towards that shown in Fig. 6.2. Finally, due to the non-deterministic nature of the solver, the number of solutions stored may differ when running

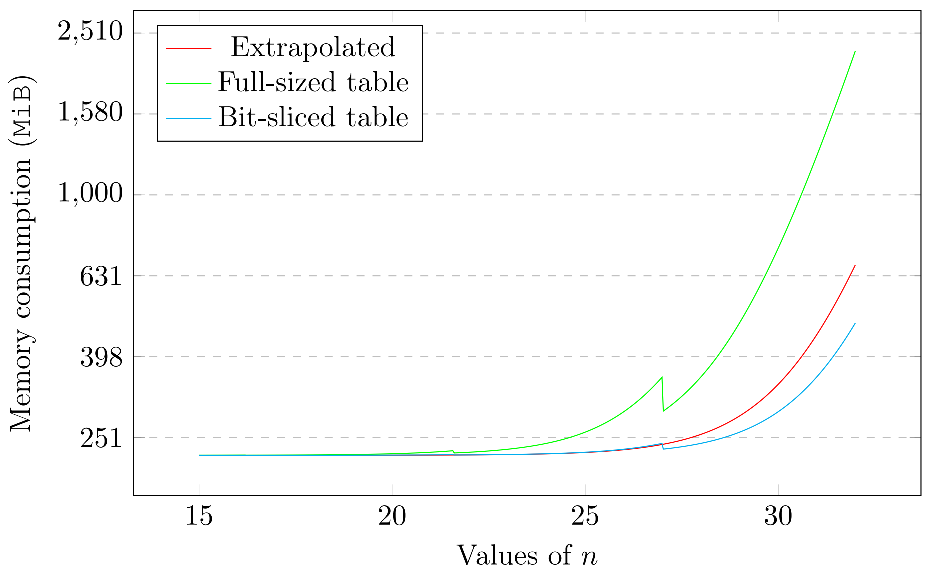

Figure 6.2: Three strategies for storing solutions, where one curve was fitted from actual memory consumption data on systems from $n = m = 20$ to $n = m = 30$. The two others are purely based on theoretically expected values. Memory data is sampled from `/proc/<pid>/status`. The red curve was fitted from *actual* memory consumption.

the solver on one system multiple times. Expectedly, no drastic changes would occur with high frequency, but it is still a risk.

6.1.2 SIMD

As already explained, one of the key goals for the vectorized implementation was to better handle memory consumption. By storing multiple systems $\tilde{\mathcal{P}}_k^{(i)}$ (for $k = 0, \ldots 3$) the solution processing and checking can be handled simultaneously inside the interpolation/evaluation procedure `fes_recover_vectorized()`. This way, no solution needs to be stored other than the one currently being processed by `fes_recover_vectorized()`, as the vectorization allows for computing multiple solutions in parallel.

The fact that solutions are not stored in large arrays should be clear from Fig. 6.1, as both graphs (b) and (c) do not yield the same slow increase in memory as graph (a). Mostly, the two SIMD implementations seem rather close in their memory consumption, even though one uses registers of double the size. With an observant eye, it can be seen that the 256-bit AVX implementation has a larger memory consumption, however, more precise profiling of the memory usage would be needed to say if it is exactly double. Of course, the memory usage as a whole is not expected to be double, as the two implementations mostly store non-vectorized elements the same. Also, recall that the SageMath front-end sets a strong baseline in memory consumption before any C procedures are called.

For a closer inspection of the memory consumption when compared to the standard

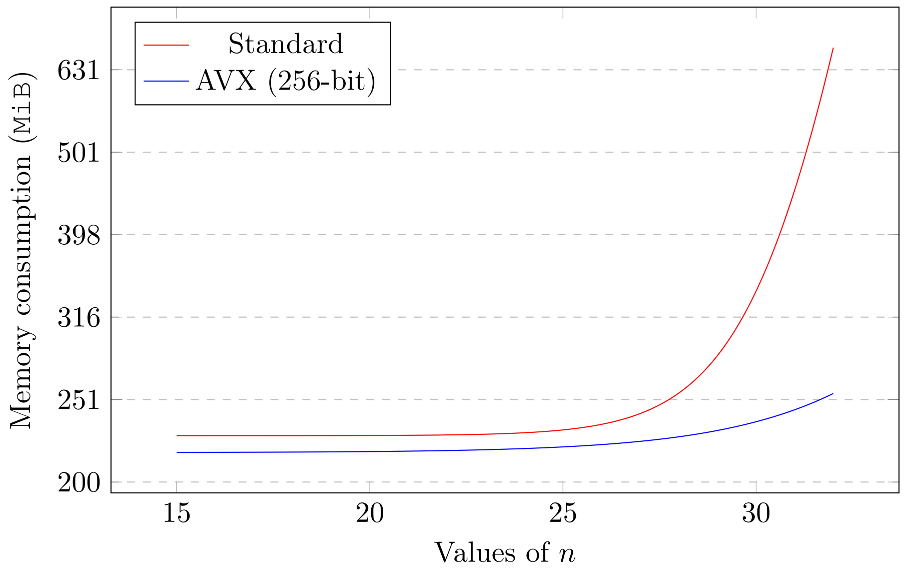

Figure 6.3: Memory consumption plotted as functions of n for the *standard* and *vectorized* implementations. The *standard* curve is the same as the extrapolated curve in Fig. 6.2. Memory data is sampled from `/proc/<pid>/status`.

implementation, observe Fig. 6.3. The two curves are sampled from different data points gathered from 5 runs of 11 different parameter sets. Now, as explained in Section 5.4, the memory consumption of the vectorized implementation still acts like an exponential function in n, due to the relationship between w and n. Even though both implementations store exponential amounts of memory, the vectorized implementation has drastically slower growth, as Fig. 6.3 clearly shows. This exponential growth stems from the storage of the derivative table in `fes_recover_vectorized()`, which holds $\sum_{i=0}^{w+1} \binom{n-n_1}{i}$ vector and is reset each round.

The expected memory consumption for the derivative table alone is at most

$$32 \cdot \sum_{i=0}^{\lceil \frac{n}{5.4} \rceil + 3} \binom{n - n_1}{i}$$

bytes as $w \leq 2(n_1+1) - n_1$ when solving quadratic systems. By using Fig. 6.1 as a reference, this would yield a total upper bound of 3 918 016 bytes, or ≈ 4 `MiB`.

Typically, the storage of AVX vectors in main memory requires specifically aligned memory, as alternatively specialized instructions (with worse performance) are needed. This alignment, 32 bytes for 256-bit vectors and 16 bytes for 128-bit vectors should be taken into consideration when comparing the theoretical 4 `MiB` of storage to what is displayed in Fig. 6.1. An interesting element of graphs (b) and (c) is the sudden increase in memory consumption early on in the procedure. Since the vectorized implementations fix n_p variables and store 2^{n_p} new system of $n - n_p$ variables, the memory build-up is expected quite large

Compile config	Solve time	Total recovery time	History checking
64	$2.86 \cdot 10^5$ ms	$2.41 \cdot 10^5$ ms	22,831 ms
256	13,648 ms	13,647 ms	-
FES	8,245 ms	-	-

Table 6.1: A comparison of different procedures solving a system of $n = m = 30$. The timings are averages from 10 different systems of this size.

early on. Furthermore, the storage of derivative tables (exponentially sized) also occurs soon after variables have been fixed, and so the memory increase seen in those regions of (b) and (c) is probably heavily attributed to these storage solutions.

Comparing graphs (b) and (c) (Fig. 6.1) to graph (a), a strength of the vectorized implementations is how the memory consumption essentially reaches a static maximum. Once `fes_recover_vectorized()` has allocated derivative tables and other memory areas, no real increase happens. This solution is therefore more predictable in the consumed memory. The vectorized implementations neither risk storing many solutions in the extreme case where the solver has to spend a lot more than four rounds to find a solution. Of course, the trade-off of the vectorized approach is still the limit of *only* storing four rounds worth of values for each fixed system, possibly not catching some candidates that would otherwise have been caught.

The general trend seen from Fig. 6.1 and Fig. 6.3 is that the vectorized implementation comparably manages to tackle the memory issues of Dinur's algorithm more efficiently. As will be seen in Section 6.2, considering timings does not hurt the vectorized implementation either.

6.2 Time

In this subsection, the performance of the different implementations is compared against that of a similarly optimized FES implementation for quadratic polynomials. Here "similarly optimized" means that the procedure is optimized in the same ways as that of `fes_-eval_parity()`, the underlying FES-procedure for `fes_recover()`, `fes_recover_vvectorized()`, and Dinur's `BRUTEFORCE()` method (see Algorithm 8). This section will also do internal comparisons between the vectorized/SIMD and standard implementation of Dinur's algorithm.

6.2.1 Standard and vectorized versions

With the current implementations, both vectorized and non-vectorized, the algorithm did not beat the FES implementation in `src/c/standard/fes.c` on samples of smaller systems. The implementation(s) were not tested on systems larger than $n = m = 48$, however,

Standard	Vectorized	FES	n, m
$2.86 \cdot 10^5$ ms	13,648 ms	8,245 ms	30
$1.17 \cdot 10^5$ ms	6,799 ms	4,197 ms	29
66,790 ms	2,960 ms	2,067 ms	28
71,600 ms	1,546 ms	1,039 ms	27
36,389 ms	1,588 ms	515 ms	26
17,757 ms	517 ms	256 ms	25
8,600 ms	326 ms	134 ms	24
3,740 ms	156 ms	65 ms	23
2,618 ms	148 ms	33 ms	22
2,156 ms	73 ms	16 ms	21
1,208 ms	44 ms	8 ms	20

Table 6.2: A comparison of total solve times for different procedures solving different system sizes. The timings are averages from 11 different systems of this size.

the vectorized version was run using a multicore implementation on such a system specifically (see Section 6.3).

Consider Table 6.1, which shows average timings for solving 11 different systems of $n = m = 30$ using the three algorithms of interest. In these tests, the vectorized implementation obtains approximately a 20 times speedup over the non-vectorized implementation in total solve time. In Table 6.2 much the same picture is drawn. The speedup from the standard version to the vectorized version is not a perfect 20 times at each parameter-set, however, the general trend is that at least a 20 times speedup is seen. Again, these measurements are not a perfect science but do show a trend.

An interesting characteristic of the timings of the standard version is that the increase in solve time seems to be *slower* in some parameter ranges. For instance, at $n = m \in \{27, 28\}$ and $n = m \in \{21, 22\}$ (Table 6.2) the increase in time is either very slow or upright negative. Examining the behavior of the function that determines n_1, i.e. $n_1 = \lceil \frac{n}{5.4} \rceil$, the *stepping* nature of the ceiling function shows directly in the timings. For both $n = m \in \{27, 28\}$ and $n = m \in \{21, 22\}$, the `fes_recover()` (and `fes_recover_vectorized()`) procedure (being the dominating contributor to timings) iterates through 2^{17} and 2^{22} values of the $\mathbf{y}$-bits, respectively. In other cases, the increase is essentially exponential. This would of course have looked different, had n_1 been computed differently.

By fitting exponential curves onto the data from Table 6.2 one obtains Fig. 6.4 (and Fig. 6.5). From these curves, the general trend from the timings of Table 6.2 are shown more clearly. As a function of the number of variables, n, the solve times for the *standard* implementation tend to grow a lot quicker than those of the vectorized implementation. As can be seen in Table 6.1, a large contributor to the overall solve time is the history-checking phase. In Section 6.1, the amount of stored candidate solutions was disclosed as being a lot

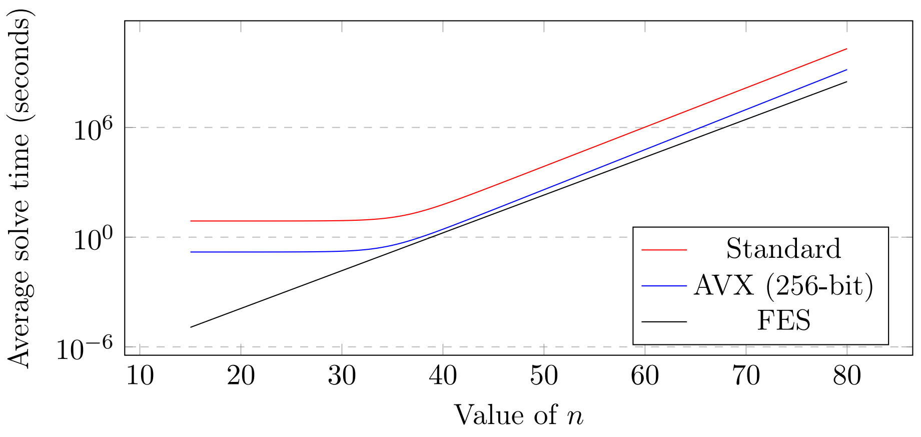

Figure 6.4: A plot of the fitted curves for all three procedures, expressing *solve*-time as a function of n. The curves are fitted from actual timings, specifically the samples of Table 6.2.

more than theoretically expected. Since timings for history-checking are directly correlated to how many candidate solutions are stored, a non-negligible amount of time could be saved.

Further, the largest contributor to solve times for both implementations of the algorithm is the FES-based recovery. This is expected, given that this procedure replaces the Möbius transform interpolation and evaluation, both being large contributors to the complexity of the original algorithm. With Table 6.3 the distribution of time spent evaluating and interpolating can be read, as well as the overall recovery time. Of course, this recovery time also includes checking evaluations for $U_0(\hat{\mathbf{y}}) = 1$, and household tasks like memory allocation. These contributions do affect the timings quite a lot, but for now, the thesis will focus on individual timings for evaluation and interpolation.

From the samples in Table 6.3, larger parameter sizes seem to change the balance of interpolation and evaluation. In the lower rows of the table, interpolation is the slowest part but in the upper rows, the roles are reversed. It was expected that interpolation would be the slowest of the two, as this included both running `fes_eval_parity()` internally, as well as interpolating derivative table entries. Examining the evaluation part of `fes_recover()` (and `fes_recover_vectorized()`), many operations are rather simple, except indexing. It turns out that the indexing operation is quite a heavy operation when computed as in Listing 5.3.15. This computation is also performed often, and as $n-n_1$ increases, the number of evaluations expectedly increase. The heaviest operation of the evaluation is indexing and must for this reason also act as a large contributor to interpolation times, especially on smaller parameter sizes. To be more clear, as n increases, more vectors in $\{0,1\}^{n-n_1}$ will have a hamming-weight greater than `deg`, triggering evaluation instead of interpolation (see Algorithm 9).

Applying well-known optimization techniques like loop-unrolling could help boost the performance of the FES-based recovery procedure. Unrolling the for-loops inside the evaluation and interpolation phases will make the indexing operations faster, as many indices then

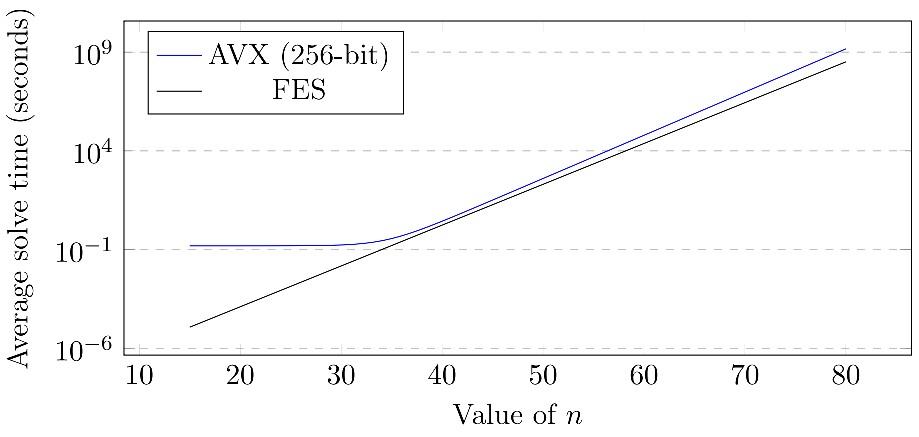

Figure 6.5: A zoom-in on the curves for the vectorized implementation and basic FES implementation, expressing *solve*-time as a function of n. The curves are fitted from actual timings, specifically the samples of Table 6.2.

become constant. This would be an interesting addition to these performance measures.

6.2.2 Comparing against FES

For the sake of comparison, both the standard and vectorized implementation were compared to an FES implementation for quadratic systems. As already explained, this implementation was optimized in pretty much the same way as `fes_eval_parity()` in `src/c/standard/fes.c`. A comparison of the total solve times can once more be found in Table 6.2. With an almost perfect doubling from $n = m = 20$ to $n = m = 30$, the FES procedure manages to outperform even the vectorized implementation of Dinur's algorithm. As is depicted by the fitted exponential curves in Fig. 6.5 and data of Table 6.3, the simple FES implementation manages to outperform the vectorized implementation by around a factor of two, quite consistently. Comparing the standard implementation to the FES implementation, the performance speedup of FES is downright distressing for the standard implementation of Dinur's algorithm. Using the extrapolations of Fig. 6.4, no crossover point can be found for which FES would become slower than Dinur's algorithm.

Many reasons may exist for these gloomy performance results when compared to FES, a few of which are already disclosed in earlier sections. Nonetheless, the comparison is still important as it shows the reality of these implementations. Although many time contributors (in practice) were disclosed in former sections, the analysis of [9] does make certain assumptions. First off, the paper focuses on cryptographically relevant parameter ranges, which the samples from Table 6.2 are not even near. Also, even though the complexity measure is *bit operations* (and non-asymptotic), certain operations are declared negligible and thereby ignored for the total amount of bit operations in the analysis. As the tests ran here are classified as smaller systems (in 2023 at least), these *negligible* performance contributors

Recovery	Interpolation	Evaluation	n, m
$2.41 \cdot 10^5$ ms	53,864 ms	98,575 ms	30
$1.01 \cdot 10^5$ ms	28,329 ms	37,255 ms	29
57,442 ms	19,667 ms	18,624 ms	28
59,433 ms	12,145 ms	24,750 ms	27
30,625 ms	7,970 ms	11,482 ms	26
15,064 ms	4,886 ms	4,965 ms	25
7,445 ms	2,923 ms	2,095 ms	24
3,256 ms	1,501 ms	751 ms	23
2,281 ms	1,202 ms	411 ms	22
1,834 ms	692 ms	516 ms	21
1,029 ms	463 ms	234 ms	20

Table 6.3: A comparison of FES-based recovery timings (non-vectorized), and internal timings, for different system sizes. The timings are averages from 11 different systems of this size.

may have had relevance for the data sampled and curves being fitted. An example of a *negligible* term in the analysis of [9] is the interpolation of polynomials using the Möbius transform (when compared to the `BRUTEFORCE()` procedure of Algorithm 8). In the data sampled in Table 6.3, although only for the standard implementation, the interpolation of polynomials holds a significant proportion of the total recovery time. Of course, comparing the theoretical analysis to practically sampled data is not a one-to-one mapping for many reasons. One such reason is that the analysis is based on the use of the Möbius transform, and the samples are based on an FES-based recovery approach (Chapter 4).

6.3 Solving large systems

Using the data from Table 6.2, and thereby curves of Fig. 6.4, to extrapolate the time it would take to solve a large system provides rather pessimistic results. For a 48-variable system in 48 polynomials, the authors of [3] found all common zeros in 21 minutes using a GPU. By extrapolating the time to solve a 48-variable system (in 48 polynomials) using the sampled data from the former sections, the result is around 1816 hours or 75 days. This extrapolation is from the data on the vectorized implementation, i.e. the fastest of the two C implementations. Of course, it could be argued that more samples would be needed to provide more accurate predictions, however, this would likely not show competitive results (compared to established solvers like FES).

Instead of trying to solve a system for 75 days on a single core, a 48-variable system in 48 equations was attempted solved on a multicore machine. The attempt was run on a 128-core (256-thread) x86-based machine. The vectorized implementation using 256-bit

AVX registers, running in parallel on 256 threads, managed to solve such a system in about 25 hours. Recall from Section 5.4 that running on a multicore system, the procedure fixes $n_p = \log_2 q$ variables on q core machines. This meant that $\log_2 256 = 8$ variables were fixed before handing the 2^8 new systems to individual CPU threads.

Now, compare the solve-time to the single-core AVX (again, 256-bit vectors) implementations by, once more, extrapolating the solve times. In a perfectly scaled world, the total solve time of the multicore run would match solving an $n = m = 48 - 8 = 40$-sized system. Extrapolating the single-core timings, a system of $n = 40$ would be solved in about 5 hours. Instead, a system of $n = 42$ would be solved in about 23 hours, according to the extrapolated data, being more comparable to the 25 hours of the multicore version on $n = m = 48$. Of course, neither the samples nor the world, are perfect. The multicore implementation is handled by Python/SageMath and the `multiprocesing` library, meaning that the fixing of variables and bit-slicing is handled here as well. Having the overhead of Python/SageMath on fixing and bit-slicing could prove part guilty for this non-perfect scaling.

Chapter 7

Conclusion

This thesis aimed to implement and identify key performance factors in the polynomial-method algorithm of [9]. In that paper, the algorithm is advertised as having exponential speedup over exhaustive search, making it an interesting subject for practical implementations. The path between theory and practice can at times be extensive, and therefore a part of the *evaluation*-goal of the thesis was the identification of strongly theoretical areas that need extra attention when implemented in practice.

One area of interest was the interpolation and evaluation of polynomials, used in one of the sub-procedures of [9]. As an alternative to using the Möbius transform, an algorithm based on the principles of the Fast Exhaustive Search procedure (FES) from [3] was introduced and implemented (Chapter 4). Instead of pursuing a variant of Dinur's algorithm using the Möbius transform, this new variant was used instead. Choosing to go with an alternate variant of Dinur's algorithm was based on the fact that many FES implementations have seen the light of day, giving more references for real implementations than when compared to the Möbius transform. Of course, the Möbius transform has seen implementations from both [2] and [14], however, Dinur's algorithm required a modified variant with less reference material to go by.

Another effect of choosing the FES-based approach for interpolation and evaluation was to better enable optimizations that would make use of the resources present in modern CPUs. One such optimization uses SIMD instructions through the AVX instruction set(s). This led to the codebase of the thesis being divided into three parts: (I) A SageMath prototype for testing the algorithmic aspects of [9]. (II) A so-called *standard* implementation in C of [9], that sought to do some optimizations but kept its relation to the reference material clear. (III) An implementation in C using AVX instructions and registers, trying to optimize as much as the thesis deadline allowed. The two C implementations were also both able to run on multicore systems, using the included Python script to run the solver.

The key points for performance evaluations are timing and memory. Especially memory consumption was proven to be extensive in the original paper [9]. The comparisons found that certain memory optimizations of the C implementations were quite welcome and certain cases saw a factor four decrease in memory consumption, from the theoretically derived values to the practically measured ones. These findings also showed that certain theoretic bounds did not hold in the implementations, in terms of the number of solutions that needed to be stored. An even stronger memory optimization was found via the vectorized AVX implementation, as this implementation managed to mitigate storing large amounts of potential solutions by implementing a lock-step parallel approach. At last, the implementations were compared to a simple implementation of the barebones FES procedure for quadratic systems. This comparison strengthened the reasons for trying to further optimize the implementation, in order to be competitive in real-world scenarios, and claim the speedup it obtained in theoretical analyses.

In general, although the practical findings of this thesis did not prove as optimistic as the theoretical ones in [9], they did show some relevant areas. This should enable future implementations, be they extensions to this or standalone, to more easily identify problems and their solutions that stem from converting theory to practice. Some of these elements can also be used for related polynomial-method algorithms that [9] used for inspiration.

Future work. In order to make the implementations from this thesis more competitive, multiple areas may be looked into. Of course, a direct comparison between an optimized Möbius transform implementation and the FES-based recovery explained in Chapter 4 would be necessary. On this note, comparing and implementing the *memory-efficient Möbius transform* would also be of great interest. Although the internal FES implementation, i.e. `fes_eval_parity()`, was optimized to some extent it was never near the likes of [3] and [4]. Therefore, optimizing this procedure more to the likes of those papers, possibly alongside the use of *monotonic Gray codes*, would yield better data on how the algorithm performs in reality.

As an alternative to implementing home-grown Möbius transforms and FES procedures, a library was published alongside [2]. It would be interesting to see this library work in tandem with the rest of a codebase like the one in this thesis, although the library may not be as specialized as the implementations from [3] and [4].

Now, the rest of the codebases for this thesis are neither perfect. Some future work should be dedicated to understanding why such a large deviation in stored candidate solutions occur (mentioned in Section 6.1), compared to the theoretically expected amounts. This would of course also help increase performance for both C variants, while minimizing memory usage. In terms of timings, alternatives for the indexing operation of `fes_recover()` and `fes_recover_vectorized()` could also be interesting, as this helps bring down the overall time spent on FES-based recovery.

For parallelization, altering the multicore handling to be exclusively C would likely help a lot too. This way, using a library like OpenMP[1], the fixation of variables could be handled at once, w.r.t. the vectorized implementation. Currently, the fixation of variables occurs at two levels, once when multiple cores need an independent system and once when filling entries in the AVX vectors. On this note, introducing larger integer registers for the standard implementation, or more compactly packing systems into AVX registers, could also be an

[1] `https://www.openmp.org/`

interesting addition. Currently, the vectorized implementation risks wasting a non-negligible amount of bits. E.g. for $\ell = 9$, the vectorized implementation (using 256-bit registers) would store $\tilde{\mathcal{P}}_k^{(i)}$s in 16 16-bit integers in the 256-bit AVX register where each integer only actually uses 9 bits. By compacting these $\tilde{\mathcal{P}}_k^{(i)}$s more tightly in the vector register, the implementation may be able to store more rounds or fixed systems. This would also allow for a replacement of unsigned integers as the underlying data structure, for bit-slicing polynomials in the standard implementation.

Bibliography

[1] Ward Beullens. „Breaking Rainbow Takes a Weekend on a Laptop“. In: Springer-Verlag, 2022.

[2] Charles Bouillaguet. *Boolean Polynomial Evaluation for the Masses.* Cryptology ePrint Archive, Paper 2022/1412. 2022. URL: https://eprint.iacr.org/2022/1412.

[3] 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: *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop.* Vol. 6225. Lecture Notes in Computer Science. Springer, 2010, pp. 203–218. DOI: 10.1007/978-3-642-15031-9_14. URL: https://www.iacr.org/archive/ches2010/62250195/62250195.pdf.

[4] 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 — Extended Version“. In: (2013). URL: https://eprint.iacr.org/2013/436.

[5] Tung Chou. „Fast Exhaustive Search for Polynomial Systems over $\mathbb{F}_2$“. MA thesis. National Taiwan University, 2010.

[6] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. „Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations“. In: *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding.* Vol. 1807. Lecture Notes in Computer Science. Springer, 2000, pp. 392–407. DOI: 10.1007/3-540-45539-6_27. URL: https://www.iacr.org/archive/eurocrypt2000/1807/18070398-new.pdf.

[7] Nicolas T. Courtois and Josef Pieprzyk. „Cryptanalysis of block ciphers with overdefined systems of equations“. In: *Advances in Cryptology—ASIACRYPT 2002.* Springer. 2002, pp. 267–287.

[8] Jintai Ding, Albrecht Petzoldt, and Dieter S. Schmidt. „Multivariate Cryptography“. In: *Multivariate Public Key Cryptosystems.* Springer US, 2020, pp. 7–23. ISBN: 978-1-0716-0987-3. URL: https://doi.org/10.1007/978-1-0716-0987-3_2.

[9] Itai Dinur. „Cryptanalytic Applications of the Polynomial Method for Solving Multivariate Equation Systems over GF(2)“. In: Springer-Verlag, 2021. DOI: 10.1007/978-3-030-77870-5_14.

[10] Itai Dinur and Adi Shamir. „An Improved Algebraic Attack on Hamsi-256“. In: *Fast Software Encryption - 18th International Workshop, FSE 2011*. Vol. 6733. Lecture Notes in Computer Science. Springer, 2011, p. 88. DOI: 10.1007/978-3-642-21702-9_6. URL: https://www.iacr.org/archive/fse2011/67330090/67330090.pdf.

[11] 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*. New York, NY, USA: Association for Computing Machinery, 2002, 75–83. ISBN: 1581134843. URL: https://doi.org/10.1145/780506.780516.

[12] Jean-Charles Faugére. „A new efficient algorithm for computing Gröbner bases (F4)“. In: *Journal of Pure and Applied Algebra* 139 (1999), pp. 61–88. URL: https://www.sciencedirect.com/science/article/pii/S0022404999000055.

[13] *Intel® Intrinsics Guide*. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html. Accessed: 2023-01-20.

[14] Antoine Joux. *Algorithmic Cryptanalysis*. Chapman & Hall/CRC Cryptography and Network Security Series. Taylor & Francis, 2009. ISBN: 9781420070026.

[15] Antoine Joux and Vanessa Vitse. *A crossbred algorithm for solving Boolean polynomial systems*. 2017. URL: https://eprint.iacr.org/2017/372.

[16] Daniel Lokshtanov, Ramamohan Paturi, Suguru Tamaki, Ryan Williams, and Huacheng Yu. „Beating Brute Force for Systems of Polynomial Equations over Finite Fields“. In: *Proceedings of the 2017 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 2190–2202. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611974782.143.

[17] Sean Murphy and Matthew J. B. Robshaw. „Essential Algebraic Structure within the AES“. In: *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*. Vol. 2442. Lecture Notes in Computer Science. Springer, 2002, pp. 1–16. DOI: 10.1007/3-540-45708-9_1. URL: https://iacr.org/archive/crypto2002/24420001/24420001.pdf.

[18] Harris Nover. „Algebraic cryptanalysis of AES: an overview“. In: (2005).

[19] *Quantum-centric supercomputing: The next wave of computing*. Accessed: 2023-01-20. URL: https://research.ibm.com/blog/next-wave-quantum-centric-supercomputing.

[20] Peter W. Shor. „Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer“. In: *SIAM Journal on Computing* 26.5 (), pp. 1484–1509. URL: `https://doi.org/10.1137\%2Fs0097539795293172`.

[21] Marion Videau. „Cube Attack“. In: *Encyclopedia of Cryptography and Security*. Springer US, 2011, pp. 289–290. ISBN: 978-1-4419-5906-5. URL: `https://doi.org/10.1007/978-1-4419-5906-5_342`.

[22] Richard Ryan Williams. „The Polynomial Method in Circuit Complexity Applied to Algorithm Design (Invited Talk)“. In: *Foundations of Software Technology and Theoretical Computer Science*. 2014.