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:
Assistant Professor Ruben Niederhagen

Author:
Mikkel Juul Vestergaard

Academic Year 2022/2023

Resumé

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

1 Introduction (approx. 3 pages)

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 lie as a key building block for enabling such properties on modern technologies. **(You can make this sentence active very easily: "Peter Shor proposed an algorothm...")** In 1994, an algorithm, due to Peter Shor [14], was proposed 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.

(Langauge is a bit 'rich' here...) With an eye on what a computing device based on quantum mechanics could further provide modern societies **(Example?)**, 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 [13]. 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*, 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, NIST called for a fourth submission round, asking for cryptographic schemes not based on lattice problems [1]. **(Strictly speaking, there is a fourth round with remaining schemes that had been submitted already in 2017. Additionally, NIST plans an on-ramp of additional signature schemes this June.)**

(This paragraph starts very abruptly; maybe introduce the four main PQC families first and then focus on MQ?) One family of post-quantum cryptosystems is that of multivariate cryptography. This branch of cryptography revolves around two mathematical problems, the MQ problem, and the IP problem. The central problem is that of the MQ problem [7].

Definition 1 (The MQ Problem)**.** Given a system of m **(Give an intuition and an example and define.)** multivariate quadratic polynomials $\mathcal{P} = \{p^{(1)}, \ldots, p^{(m)}\}$, over the ring of polynomials in n variables $\mathbb{F}[x_1, \ldots, x_n]$, find values $\bar{\mathbf{x}} = (\bar{x}_1, \ldots, \bar{x}_n)$ that satisfy

$$p^{(1)}(\bar{\mathbf{x}}) = p^{(2)}(\bar{\mathbf{x}}) = \cdots = p^{(m)}(\bar{\mathbf{x}}) = 0$$

Here, $\mathbb{F} = \mathbb{F}_q$ is a finite field of q elements. **PUT IN PREREQUISITES?** **(Maybe use an intuitive notion here and move the formal definition to Section 2.)**

Choosing quadratic polynomials over other degrees of polynomials is often a product of efficiency.

(How hard is the MP (and the MQ) problem?)

(Clarify.) Given the reliance on the MQ problem in post-quantum cryptography, having more *efficient* solvers for these systems (MQ solvers) will directly impact the security of multivariate cryptographic schemes. Using existing and well-known solvers, the NIST candidate scheme `Rainbow` was broken by Ward Beullens in [2], 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 with 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 [12]. The first attack, due to Courtois and Pierprzyk [6], 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 [11], 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 bruteforce procedure.

Other examples of algebraic cryptanalysis are general attacks like *cube attacks* [15] and importantly also attacks on cryptographic hash functions like that of [9]. The use-cases for efficient MQ-solvers are therefore vast and an important tool in the cryptanalysis of not only PQC schemes, but also ordinary symmetric ciphers and cryptographic hash functions. **Possibly reformulate this and "ordinary ciphers".**

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

2 Prerequisites (approx. 10-15 pages)

2.1 Polynomial method for solving MQ systems

2.2 Fast exhaustive search for multivariate polynomials

The fast exhaustive search procedure for polynomials over $\mathbb{F}_2$, [3, 4], 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 can be implemented nicely in practice, with good use of the parallelization resources present in modern computers. The algorithm needs $2d \cdot \log_2 n \cdot 2^n$ bit operations (in expectation) for systems of quadratic polynomials, and is a core element in Dinur's polynomial-method algorithm from [8].

The main FES procedure is shown in Algorithm 2 with the helper functions Algorithm 3 and Algorithm 1. This merely serves to give context to the following subsections.

2.2.1 Gray codes

Revisit

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 encoding 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 encoding 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, 4] is not the only one, as multiple other types with different additional properties exist.

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 (being $1_2, 0_2$) and *appends* it to the original sequence. Lastly, one prepends 0 to entries of the first half of this new sequence and a 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.

From the construction method just explained, it can be derived that constructing the ith codeword, g_i can be done using the formula

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

where $>>$ denotes a logical right-shift operation in the binary representation of i and $\oplus$ denotes the bit-wise *xor* operation between two binary numbers. This formula can be derived by observing that inverting the bit at position i of all binary encoded numbers in the sequence $0, \dots 2^n - 1$ will change the order of blocks of 2^i codewords.

2.2.2 Exhaustive Search using Gray Codes

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

Definition 3 (Derivatives)**.** Let $\{\mathbf{e}_0, \dots, \mathbf{e}_{n-1}\}$ denote the canonical basis over the vector-space $(\mathbb{F}_2)^n$. The derivative of a polynomial, p, in the ring $\mathbb{F}_2[x_0, \dots, x_{n-1}]$ w.r.t. the ith variable is $\frac{\partial p}{\partial x_i} : \mathbf{x} \mapsto p(\mathbf{x} + \mathbf{e}_i) + p(\mathbf{x})$.

In order to minimize the amount of operations needed between iterations in an exhaustive search procedure, the authors of [3] suggests to look at inputs in gray code order. Examining inputs in gray code order allows for efficient use of partial derivatives for computing the output of one input based on the evaluation of the previous input. Inspecting Definition 3 reveals the foundation for 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 consecutive inputs 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 $p(\mathbf{x}_i)$ 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 of the FES procedure, or alternatively 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 the previous 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. However, stopping here leaves us with 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 $s = 10$ and $s = 11$, the algorithm stores the latest $\frac{\partial p}{\partial x_i}(\mathbf{x})$ that has been computed and ensures to update it by recursively looking at the only variable, x_j, that changed since last time x_i toggled. This means that we may update $\frac{\partial f}{\partial x_i}(\mathbf{x})$ by adding $\frac{\partial^2 p}{\partial x_i \partial x_j}(\mathbf{x})$ to the stored value. For quadratic polynomials, this second derivative would be

```
Algorithm 1: STEP(state)
  Input: state
1 $state.i \leftarrow state.i + 1$
2 $k1 \leftarrow \mathrm{BIT}_1$(state.i)
3 $k2 \leftarrow \mathrm{BIT}_2$(state.i)
4 if k2 exists in state.i then
5     $s.d'[k_1] \leftarrow s.d'[k1] \oplus s.d''[k1, k2]$
6 end
7 $s.y \leftarrow s.y \oplus s.d'[k1]$
```

Figure 1: Step

```
Algorithm 2: 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: Eval

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

Figure 3: INIT

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 binary encoding of $\mathbf{x}_i$ we have that the FES procedure looks not only for $b_1(\mathbf{v}_i)$ (Definition 2) but $b_k(\mathbf{v}_i)$ for $k = 1, \dots, d$. For the quadratic case in Fig. 1 we see the procedures BIT_1 and BIT_2 representing $b_1(\mathbf{x}_i)$ and $b_2(\mathbf{x}_i)$ with `state.i` representing $\mathbf{v}_i$. The fact that we can simply use the binary encoding to find which bits are turned on and off 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 reside in Algorithm 1, 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 position that so far has been untouched. Therefore, some time is spent for pre-evaluating these derivative values directly. For the quadratic case, the pre-evaluation is done in Algorithm 3, lines 10-13, equivalently the valued stored is $\frac{\partial p}{\partial x_k \partial x_{k-1}} + a_k$ for a_k being the coefficient to the monomial x_k. 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 Fig. 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 p. 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 Fig. 3 to be bit-vectors instead of a singular bit then yields a version of the recursive procedure that can find all common zeroes of a system in $O(2m2^n)$ bit operations. For Algorithm 3, Algorithm 2 and Algorithm 1, the pseudo-code only represents FES on single polynomials, but should still be clear how to parallelize using bit-vectors and inputting entire systems $\mathcal{P}$ (with appropriate method calls) instead of single polynomials p.

2.2.3 Early abort and Naive evaluation

The complexity of $O(2m2^n)$ bit operations is neither what the FES authors claim, [3, 4], nor what Dinur necessitates, [8]. Using the procedure described earlier, further ideas may be used for a more efficient exhaustive search. Using the parallelization already mentioned, the authors of [3, 4] note that using an early abort strategy alongside the recursive algorithm already mentioned yields an algorithm which finds all common zeroes of m polynomials in $log_2 n \cdot 2^{n+2}$ bit operations. This early abort strategy is essentially to compute the common zeroes of k polynomials (with a well-chosen k) in parallel, followed by then sequentially computing the common zeroes of the remainder of the polynomials. This last part is what [3, 4] denotes as *candidate checking* as it boils down to checking only the common zeroes of the first k polynomials, as any other evaluation point trivially cannot be a common zero of all m polynomials.

2.2.4 Partial evaluation and FES

Two important ideas for obtaining a sufficient parallelization of the combined procedure (in practice) from Section 2.2.2 and Section 2.2.3; 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.

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 2.2.2. 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 [8].

2.3 Polynomial interpolation

2.4 Dinur's polynomial-method algorithm

A specific instance of the polynomial-method type algorithms for solving multivariate quadratic polynomial systems (see Section 2.1) is the algorithm of [8], due to Dinur. As the title of the thesis states, this will be the algorithm in focus in the following sections.

2.4.1 Complexities

With the concrete complexities of the algorithms in [10, 16] being larger than 2^n, a concretely efficient algorithm for cryptographic purposes was yet to be seen prior to [8]. In 2021, Dinur formulated a polynomial-method based algorithm with the purpose of being applicable in cryptography in general, and specifically for cryptanalytic purposes. This meant that the non-asymptotatic complexities ought to be good even for very large problem sizes, due to the natural parameter sizes in cryptology. The asymptotic complexities of the formerly mentioned algorithms of [10, 16] may therefore be better, while in a non-asymptotatic context not yielding the exponential speedup over exhaustive search as advertised. The algorithm provided in [8] therefore has the interesting property of yielding exponential speedup over exhaustive search, even for very large problem sizes. In this vein, the algorithms of [10, 16] has been revealed to have a concrete complexity larger than 2^n for cryptography-relevant parameters.

From analysis, the algorithm in [8] is bound to $n^2 \cdot 2^{0.815n}$ bit operations for systems of quadratic polynomial, 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 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 the memory usage. The spatial complexity of this algorithm is therefore also quite vast, and was shown to be reducible to around $n^2 \cdot 2^{0.063n}$ bits for quadratic polynomials systems. These complexities were proven in [8] as well.

2.4.2 The algorithm

Algorithm 4: 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      $A \leftarrow$ MATRIX($l$, $m$)
6      $\tilde{E}_k \leftarrow \{\sum_{j=0}^{m-1} A_{i,j} \cdot p_j(\mathbf{x})\}_{i=0}^{\ell-1}$
7      $w \leftarrow \tilde{E}_k$.degree()
8      $CurrPotentialSolutions \leftarrow$ OUTPUT_POTENTIALS($\tilde{E}_k$, $n1$, $w$)
9      $PotentialSolutions[k] \leftarrow CurrPotentialSolutions$
10     foreach $\hat{y} \in \{0,1\}^{n-n1}$ do
11         if $CurrPotentialSolutions[\hat{y}][0] = 1$ then
12             if $CurrPotentialSolutions[\hat{y}] = PotentialSolutions[k_1][\hat{y}]$ then
13                 $sol \leftarrow \hat{y} \parallel CurrPotentialSolutions[\hat{y}]$
14                 if TEST_SOLUTION($\mathcal{P}$, $sol$) then
15                     return $sol$
16                 end
17             end
18         end
19     end
20 end
```

Algorithm 5: OUTPUT_POTENTIALS()

Input: $\tilde{E}_k$:

Algorithm 6: COMPUTE_U_VALUES()

Algorithm 7: BRUTEFORCE()

2.5 Platform and architecture

3 Extensions to the original scheme (approx. 10 pages)

3.1 Polynomial interpolation using FES

4 Implementation (approx. 15-20 pages)

The accompanying git repository contains more than one implementation, or *variant*, of Dinur's original algorithm. These variants are divided into a faster `C` implementation and a prototype `Sagemath` implementation. `C` function declarations can be found in the `inc/` folder, other code can be found under `src/`.

4.1 Sage code

As was implied earlier, the `Sage` 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 [8], in a more or less described as the pseudo code presented. The three main procedures described by Dinur can be found in `sage/dinur.sage` with some accompanying convenience and test functions. A bit-sliced version of the FES procedure, described in [3], for quadratic polynomials can be found in `sage/fes.sage`. This implementation is not as heavily optimized as those in [3] and [4], simply due to the `Sage`-induced overhead counteracting fine-adjusted optimizations. The prototype code further introduces a prototype of a FES-based recovery, acting as an alternative to the Möbius Transform originally described by Dinur. The Möbius Transform was implemented in `mob.sage` and allows for a *sparse*-transform used for interpolating the U_i-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 recovery and using the Möbius transform is a simple boolean switch in the `solve` and `output_potentials` functions in `src/dinur.sage`. **IF MOB IS ALTERED IN SAGE; CHANGE THIS**

Other than the prototype code implemented in `Sagemath`, a *front-end* was also implemented allowing for easier loading, generation and calling of the optimized `C` code. **INSERT HOW TO CALL SAGE CODE**

4.2 Core algorithms

In this subsection, the implementation of the core algorithms for Dinur's algorithm are described,

4.2.1 Fast Exhaustive Search over $\mathbb{F}_2$

4.2.2 Möbius Transform

4.2.3 Dinur's solver

4.3 Optimizations

1. Tight integration of U-value computation with polynomial interpolation and full evaluation
2. Sparse Möbius transform
3. FES-recover
4. Monotonic gray-codes for bruteforce subprocedure
5. `C`-specific optimizations
6. Handling memory
7. Concurrency

4.3.1 Fast Exhaustive Search

4.3.2 Dinur's solver

5 Evaluation (approx. 10-15 pages)

6 Conclusion (approx. 2 pages)

References

[1] Gorjan Alagic et al. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standa en. 2022-09-29 04:09:00 2022. DOI: https://doi.org/10.6028/NIST.IR.8413-upd1. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=935591.

[2] Ward Beullens. Breaking Rainbow Takes a Weekend on a Laptop. Cryptology ePrint Archive, Paper 2022/214. https://eprint.iacr.org/2022/214. 2022. URL: https://eprint.iacr.org/2022/214.

[3] Charles Bouillaguet et al. Fast Exhaustive Search for Polynomial Systems in F_2. Cryptology ePrint Archive, Paper 2010/313. https://eprint.iacr.org/2010/313. 2010. URL: https://eprint.iacr.org/2010/313.

[4] Charles Bouillaguet et al. Fast Exhaustive Search for Quadratic Systems in $\mathbb{F}_2$ on FPGAs — Extended Ve Cryptology ePrint Archive, Paper 2013/436. https://eprint.iacr.org/2013/436. 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 T Courtois and Josef Pieprzyk. „Cryptanalysis of block ciphers with overdefined systems of equations“. In: Advances in Cryptology—ASIACRYPT 2002: 8th International Conferenc Springer. 2002, pp. 267–287.

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

[8] Itai Dinur. Cryptanalytic Applications of the Polynomial Method for Solving Multivariate Equation Syste Cryptology ePrint Archive, Paper 2021/578. https://eprint.iacr.org/2021/578. 2021. URL: https://eprint.iacr.org/2021/578.

[9] Itai Dinur and Adi Shamir. „An Improved Algebraic Attack on Hamsi-256“. In: Fast Software Encryption. Ed. by Antoine Joux. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 88–106. ISBN: 978-3-642-21702-9.

[10] Daniel Lokshtanov et al. „Beating Brute Force for Systems of Polynomial Equations over Finite Fields“. In: Proceedings of the 2017 Annual ACM-SIAM Symposium on Discrete Algorithms (pp. 2190–2202. DOI: 10.1137/1.9781611974782.143. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611974782.143. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611974782.143.

[11] Sean Murphy and Matthew J.B. Robshaw. „Essential Algebraic Structure within the AES“. In: Advances in Cryptology — CRYPTO 2002. Ed. by Moti Yung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1–16. ISBN: 978-3-540-45708-4.

[12] Harris Nover. „Algebraic cryptanalysis of AES: an overview“. In: University of Wisconsin, USA (2005), pp. 1–16.

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

[14] Peter W. Shor. „Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer“. In: SIAM Journal on Computing 26.5 (Oct. 1997), pp. 1484–1509. DOI: 10.1137/s0097539795293172. URL: https://doi.org/10.1137%5C%2Fs0097539795293172.

[15] Marion Videau. „Cube Attack“. In: Encyclopedia of Cryptography and Security. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 289–290. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_342. URL: https://doi.org/10.1007/978-1-4419-5906-5_342.

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

A First appendix

B Second appendix