Radboud University

Map your Neighborhood: Assessing an algorithm for solving the Isomorphism of Polynomials problem

Faculty of Science

Author:
Lavika Singhal

Daily supervisor:
Monika Trimoska

First assessor:
Simona Samardijska

Second assessor:
Bram Westerbaan

Contents

1 Introduction

The Isomorphism of polynomials (IP) problem was described by Patrain in 1996[8] as testing the equivalence of two polynomial maps. Considering a setting in which the adversary is given two quadratic functions and the definite information that they are equal up to linear changes of coordinates the goal of the adversary is to compute these changes of coordinates. This requires finding an isomorphism between the two functions, a task which is as we will see in section 2.2, at least as difficult as the Graph Isomorphism problem (shown in[9]).

The Isomorphism of Polynomials problem is closely related to the attack of finding private keys for multivariate public key cryptosystems (section 2.2.1), for example the attack proposed by Patrain [8] on the Matsumoto-Imai cryptosystems[11], making the problem particularly relevant in cryptography.
One characteristic of the IP problem that makes it harder to analyse is that it can take various different forms depending on the parameters, such as dimensions and base field of the vector spaces.
The Quadratic Maps Linear Equivalence (QMLE) problem is a variant of the Isomorphism of Polynomials problem and the setting that appears in most cryptographic constructions. There are two cases of the QMLE problem — homogeneous and inhomogeneous. The authors of [7] observed that their algorithm the "Groebner basis" algorithm was able to solve the inhomogeneous instance of the QMLE problem in polynomial time. However, this was not the case for the homogeneous instance and is thus the problem of focus in the paper[2].

We address the challenge of finding isomorphisms between two quadratic functions by implementing the Map your Neighborhood algorithm given by the authors in [2]. The authors of [2] restrict themselves to the even characteristic of the finite field to avoid presenting two variants for their results, as the theory of quadratic forms presents differently for odd characteristic and for characteristic two. In principle the algorithm can work for the odd characteristic of the finite field, the impact that this has on the performance of the algorithm is investigated in this thesis by comparing the algorithm's performance over the even and odd characteristic of the finite field.
Concretely, the Map your Neighborhood algorithm solves the QMLE problem in the homogeneous case. The algorithm is based on the reduction of the homogeneous QMLE problem to graph isomorphism by associating each quadratic map with a corresponding graph, effectively reducing the problem of determining linear equivalence between quadratic maps into determining isomorphism between the associated graphs. This is done through the process of examining the whole connected components(group of connected vertices) in the graphs of both quadratic forms to distinguish between points with the same number of connections (degree), to identify if a pair of points exist between the quadratic maps that have an isomorphic relation between their connected components. Considering a point (represented as a vector) x from the quadratic form $\mathcal{F}$, a point y from the quadratic form $\mathcal{P}$, and the existence of a transformation matrix

S, the algorithm checks if the the relation $y = \mathbf{S}x$ or equivalently $x = y\mathbf{S}$ holds. In this thesis, x and y are implemented as row vectors, and it is checked whether the relation $x = y\mathbf{S}$ is valid.

1.1 Research questions and obtained results

The objective of this thesis is to analyse the behavior of the Map your Neighborhood algorithm in the case when the finite field is of odd characteristic, by comparing its performance for the case when $q = 2$ and $q = 3$. This helps in understanding the impact of the characteristic of the finite field on the efficiency of the algorithm and the hardness of finding potential isomorphisms between polynomials under different parameters. Which, in turn, is useful when selecting parameters to achieve the desired security of cryptosystems that are based on the IP problem.

The performance of the algorithm varies for different parameters such as the dimension of the quadratic forms represented as matrices, rank, number of vertices checked to find a collision, etc. To compare the behaviour of the algorithm between $q = 2$ and $q = 3$, the success probability, which is the ratio of the number of times a successful match is found to the total number of trials, that is, 50, is calculated in both cases under the same set of parameters. A successful match is defined as finding a pair of points (x, y) such that the relation $x = y.\mathbf{S}$ holds, within the constraint on the number of points to be checked. Additionally, to understand the impact of structural differences in the connected components of the quadratic forms in finite field of odd characteristic and characteristic two (see section 6 in [2]) on the algorithm's performance, the average number of distinct hash values generated by the subgraphs formed through the traversal of the connected components of the points is computed.
The key observation of the experimental evaluation of the Map your Neighborhood algorithm in this thesis is that the average success probability is consistently lower for $q = 3$ as compared to $q = 2$ for all sets of parameters we tested. The following graph shows this comparison for one set of parameter (matrix size):

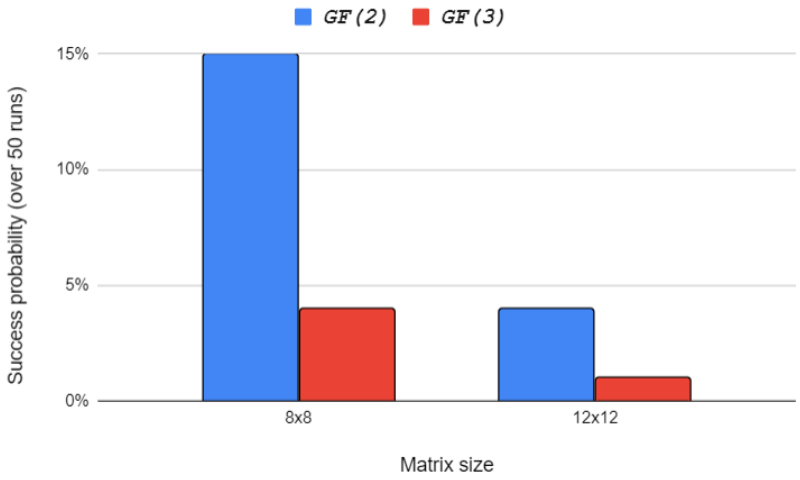


Figure 1: Average success probability of the algorithm for $q = 2$ and $q = 3$ for different matrix sizes

To analyse this difference in the behavior of the algorithm, a deeper exploration into the working of the algorithm and its experimental evaluation is discussed in detail in the subsequent sections.

1.2 Related work

The authors of [2] present three new algorithms, including the Map your Neighborhood algorithm, for solving the QMLE problem in the homogeneous case by reducing the solution of the homogeneous instance of the problem into an inhomogeneous instance through some preprocessing. The Map your Neighborhood algorithm uses the birthday paradox to improve on the time taken by exhaustive search algorithms to invert the quadratic trapdoor functions, which requires recovering the secret-key through solving an instance of QMLE. By leveraging the likelihood of finding collisions in large datasets, the algorithm improves on the exhaustive search by extracting partial information on the isomorphism such as the subgraphs formed by the connected components. This approach avoids the need for processing the exponentially large graphs derived from the quadratic maps, thereby making the problem inhomogeneous and easier to solve. The Map your Neighborhood algorithm reduces the time complexity of recovering the secret-key that is solving an instance of QMLE, to probably $\leq q^{n/2}$, while experiments indicate it can even run in $\leq q^{2n/3}$, where q^n is the time taken by exhaustive search to invert the quadratic trapdoor function[2].

Analysing this algorithm and determining its complexity is difficult due to its partially heuristic nature that arises from the use of techniques such as graph traversal, thus relying on the properties of the graphs derived from the quadratic forms. This makes it harder to formally analyse the complexity of the algorithm

as these techniques are influenced by the practical effectiveness and experimental observations, making them less deterministic theoretically. The authors conducted a thorough complexity analysis of the algorithm based on a conjecture related to random quadratic maps. Experimentally verifying that their theoretical predictions about the algorithm's behavior are not off significantly. According to their results, the success probability of the algorithm when $q = 2$ is 62%. They remarked that,"We tend to believe that the complexity and/or success probability degrade exponentially fast when q grows, but we fall short of a definitive conclusion." This sets the foundation for an investigation of the behavior of the algorithm in a finite field of characteristic other than two, and $q = 3$ is chosen in this thesis.

1.3 Organisation

This thesis is structured to explore the performance of the Map your Neighborhood algorithm in finite field of odd characteristic. We start with an introduction that outlines the relevance of the Isomorphism of Polynomials (IP) problem. Following this, the Preliminaries section (section 2) explains essential concepts, the specifics of the QMLE problem and the algorithms that currently exist for this problem. Then a detailed description of the Map your Neighborhood algorithm (section 3) is provided, focusing on its implementations including explanations of its key functions. The Experimental Results section (section 4) presents the data from the algorithm's application, focusing on its performance in $q = 2$ and $q = 3$. The interpretation of these results is given in Section 4.2. The thesis concludes by summarising the information obtained about the algorithm's performance (Section 5) and exploring directions for future research to enhance and extend the algorithm's performance and application (section 6).

2 Preliminaries

2.1 General linear group and general affine group

The general linear group of degree n is the set of $n \times n$ invertible matrices, together with the operation of matrix multiplication[4]. The general affine group of any affine space is the group of all invertible affine transformations (geometric transformations that preserve collinearity) from the space into itself. The general affine group is described as the semidirect product of the general linear group[4]. V denotes a n-dimensional vector space over the finite field $\mathbb{F}_q$[3].

2.2 The QMLE problem and its application in cryptography

The Isomorphism of polynomials(IP) problem tests the equivalence of two polynomial maps. Intuitively, these equivalences are transformations that preserve the structure of the polynomial systems. Concretely, two polynomial systems are considered equivalent if an isomorphic mapping exists between them,

as follows[10]:
Let $N \in \mathbb{N}$, $k \in \mathbb{N}$, q be a prime power, $\mathbb{F}_q$ be a finite field with q elements. $\mathcal{F}, \mathcal{P}$ denote the sets of polynomials, in particular, multivariate polynomials (polynomials with more than one variable) for this definition. $\mathrm{AGL}_n(q)$ represents the general affine group of degree n over $\mathbb{F}_q$, $\mathbf{S}$ and $\mathbf{T}$ denote the transformation matrices.

$\mathrm{IP}(N, k, \mathbb{F}_q[x_1, ..., x_N]^k \times \mathbb{F}_q[x_1, ..., x_N]^k)$:
Input: Two k-tuples of multivariate polynomials $\mathcal{F} = (f_1, f_2, ..., f_k), \mathcal{P} = (p_1, p_2, ..., p_k) \in \mathbb{F}_q[x_1, ..., x_N]^k$.

Question: Find — if any — $(\mathbf{S}, \mathbf{s}) \in \mathrm{AGL}_N(q), (\mathbf{T}, \mathbf{t}) \in \mathrm{AGL}_k(q)$ such that

$$\mathcal{P}(\mathbf{x}) = \mathcal{F}(\mathbf{xS} + \mathbf{s})\mathbf{T} + \mathbf{t}.$$

QMLE is a variant of the IP problem, where the vector space is finite (of size q), the polynomials are quadratic and the domain and co-domain are the same, that is, the polynomial maps are quadratic maps. The homogeneous case of the QMLE problem is defined in [10] as:

Let $\mathcal{F}, \mathcal{P}$ denote the sets of homogeneous multivariate polynomials (multivariate polynomials with all terms having the same degree). $\mathrm{GL}_n(q)$ represents the general linear group of degree n over $\mathbb{F}_q$, hQMLE(N, k, $\mathbb{F}_q[x_1, ..., x_N]^k \times \mathbb{F}_q[x_1, ..., x_N]^k$):
Input: Two k-tuples of homogeneous multivariate polynomials of degree 2

$$\mathcal{F} = (f_1, f_2, ..., f_k), \quad \mathcal{P} = (p_1, p_2, ..., p_k) \in \mathbb{F}_q[x_1, ..., x_N]^k$$

Question: Find —- if any —- a map ($\mathbf{S}$, $\mathbf{T}$) where $\mathbf{S} \in \mathrm{GL}_N(q)$, $\mathbf{T} \in \mathrm{GL}_k(q)$ such that

$$\mathcal{P}(x) = (\mathcal{F}(x\mathbf{S}))\mathbf{T}$$

The QMLE problem resembles the Even-Mansour cipher[6], which turns a fixed n-bit permutation P into a n-bit block cipher with a $2n$-bit key by setting $E_{k1,k2}(x) = P(x + k1) + k2$. To attack this construction, the adversary aims to recover the keys while only having black-box access to E and P. The success probability of the adversary running in time t and sending Q queries is limited by $t.Q \geq 2^n$, under the assumption that P is a random permutation[2].
The hardness of solving the QMLE problem can be leveraged for creating a similar construction in which a fixed and quadratic permutation P, is transformed into a public-key encryption function

$$E_{\mathbf{S},\mathbf{T}} = \mathbf{T} \circ P \circ \mathbf{S}$$

In this case, the adversaries are assumed to have access to both E and P, as well as how E is constructed from P using $\mathbf{S}$ and $\mathbf{T}$[2].

The main challenge of the adversary is finding the specific transformations **S** and **T**, given E and P. This problem translates into solving the QMLE problem. Hence, the security of the construction depends on the computational hardness of solving the QMLE problem.

2.2.1 Applications

The Quadratic Maps Linear Equivalence (QMLE) problem is an important equivalence problem for multivariate cryptography[10]. Multivariate cryptography is the study of public key cryptographic systems in which the trapdoor one-way functions take the form of a multivariate quadratic polynomial map over a finite field (as defined in [5]). This means that the public key is in general given by a set of quadratic polynomials and the encryption or verification process involves the evaluation of these polynomials at any given value. Inverting a multivariate quadratic map requires solving a series of quadratic equations defined over a finite field[5].

Some of the notable forms of are trapdoor-based multivariate signature systems are the Hidden Field Equations(HFE) proposed by Jacques Patrain and the Unbalanced Oil and Vinegar (UOV) scheme (built on the oil and vinegar scheme designed by Jacques Patrain [12]).
The security of such schemes relies on the inability of the attacker to efficiently solve the quadratic equations without the knowledge of the trapdoor used in generating the private key[12].

There is another type, not discussed in this paper — multivariate signatures that rely on proving the knowledge of an isomorphism. They follow the Fiat-Shamir construction and depend on the computational hardness of solving the QMLE problem.
In these systems, quadratic maps are used as the public key, while the transformations that make up the isomorphisms are kept as the secret key. This secret key is obtained by solving an instance of the QMLE problem, ensuring the security[8].

We attack the QMLE problem using graph theory and using the following concepts.

2.3 Linearity graphs

Let q be a prime power, $n \in \mathbb{N}$ and $h : (\mathbb{F}_q)^n \to (\mathbb{F}_q)^n$ a quadratic map. Then G_h is the linearity graph of h, in which the vertices are elements of $(\mathbb{F}_q)^n$, and there is an edge between the vertices x and y if and only if the following relation holds:

$$h(x + y) = h(x) + h(y)$$

The presence of an edge between vertices x and y indicates that h satisfies the additive property of linear functions for x and y.

Let $\mathbb{S}$, $\mathbb{T}$ be linear isomorphisms, $f, g : \mathbb{F}_q^n \to \mathbb{F}_q^n$ be maps. According to *Lemma* 1 in [2], given two graphs G_f and G_g, if:

$$\mathbf{T} \circ g = f \circ \mathbf{S}$$

then $\mathbf{S}$ is a graph isomorphism that sends G_f to G_g. To recover partial information about the isomorphism between G_f and G_g, it is enough to find how $\mathbb{S}$ transforms one arbitrary vertex. This is important for the purpose of the Map your Neighborhood algorithm as it allows the formulation of the problem without the presence of $\mathbf{T}$ and eliminates the need to work with exponentially large graphs.

The disconnected linearity graph G_h^* is obtained by removing the zero vertex and self-edges. This graph is in general not fully connected and contains several connected components. The dimension of the kernel of a linear map denoted as *dim ker* $D_x f$, provides information about the neighbors of the vertex x and size of the connected component that it belongs to[2]. This is discussed further in the following sections 2.4 and 2.5.

2.4 Kernel of the differential

Let f"$\mathbb{F}_q^n \to \mathbb{F}_q^n$ be a quadratic form, a non-zero vertex $x \in (\mathbb{F}_q)^n$ and $D_x f$ be the linear map, then the differential at x is given as[2]:

$$D_x f : y \to \psi(x, y) = f(x + y) - f(x) - f(y)$$

where $\psi(x, y) = f(x + y) - f(x) - f(y)$ is the polar form of the quadratic form applied to the vectors.
Computing the $D_x f$ matrix requires n matrix-vector products and determining its kernel generally takes $O(n^3)$ operations, thus finding the neighbors of a vertex x in G_f can be done in time $O(n^3)$, allowing the traversal of the vertices using a polynomial number of elementary operations in a linearity graph[2].

2.5 Dimension of the kernel of the differential

According to Lemma 2 in [2], let $x \in (\mathbb{F}_q)^n$ be a non-zero vector, and $f : (\mathbb{F}_q)^n \to (\mathbb{F}_q)^n$ be a uniformly random quadratic map. Then $D_x f$ is a uniformly random matrix vanishing over x. The probability that the differential matrix $D_x f$ has a kernel of dimension $K \geq 1$ is of the order $q^{-K(K-1)}$. Over $\mathbb{F}_q$, a K dimensional vector space contains q^K elements, so if *dim ker* $D_x f = K$, then the corresponding vertex x has q^K neighbors.
If *dim ker* $D_x f = 1$, then the only neighbors of x are its multiples, and x belongs to a connected component of size $q - 1$[2]. The dimension of the kernel of of the differential matrix of a vertex can provide information about the connectivity of the vertex. Hence, helping to reduce the complexity for finding potential isomorphisms by enabling the process of comparing connected components of

vertices with similar sizes, narrowing the search space, and increasing efficiency by reducing the number of comparisons.

2.6 Known algorithms for the QMLE problem

The following algorithms aside from the Map your Neighborhood algorithm are known for solving the QMLE problem. The Map your Neighborhood will be discussed in more detail in the next section.

The "to-and-fro" approach: This approach was proposed by the authors of the paper [9] for the QMLE problem. The working of this approach with an example and its explanation can be found in [9]. The method is called "to-and-fro" as it involves going back and forth between operations on the transformations $\mathbf{S}$ and $\mathbf{T}$, until N independent equations are derived for each of them, where N denotes the number of variables in the systems of equations.

This method exhibits variation in attack complexities based on the size of the finite field q. When $q \neq 2$, the attack complexity is higher, ranging from $N^{O(1)}O(q^N)$ to $N^{O(1)}O(q^{2N})$. This is due to the number of dependent equations generated$(q^k - 1)$ exceeding the number of independent equations(k) significantly$(q^k - 1 \gg k)$, which leads to more complexity for the computational process.

In contrast, when $q = 2$, the number of dependent equations generated$(q^k - 1)$ is similar to the number of linearly independent equations(k). Hence, reducing the attack complexity lower, given as $N^{O(1)}O(q^{2N})$.

However, this approach is only effective when the quadratic maps are bijective and does not work well for non-bijective cases according to the observations made by the authors of [9]. Different ways to adapt this approach to the non-bijective case have been found[2], but this is not the focus of the thesis.

The "Groebner basis" algorithm: The "Groebner basis" algorithm defined in [7], consists in identifying coefficient wise the equation:

$$\mathbf{T}^{-1} \circ g = f \circ \mathbf{S}$$

linking two vectors composed of N quadratic forms. Equivalently, this approach translates into solving approximately N^3 quadratic equations in the $2N^2$ coefficients that represent the unknown transformations of coordinates. These equations are solved through the computation of a Groebner basis, that is, a set of multivariate non-linear polynomials that generate a structure allowing for simple algorithmic solutions to polynomial equations.

This algorithm is well defined and methodically structured. However, the complexity of Groebner basis algorithms is difficult to analyze and the authors of the

paper [7] did not provide definitive conclusions. Although they noted through empirical observation that the algorithm concludes in polynomial time $O(n^9)$, when dealing with inhomogeneous quadratic maps.

For homogeneous cases, it was conjectured that the algorithm's complexity could be subexponential; but no concrete evidence or argument was provided to support this claim[2].

General Birthday-based Equivalence Finder algorithm: The General Birthday-based Equivalence Finder algorithm defined in [10]: Let U be a set of size N. Let ϕ be a bijection $S_1 \to S_2$ where S_1 and S_2 are subsets of ϕ; we know that ϕ exists but cannot compute it directly. Instead, we have an algorithm **FindFunction** that given $x \in S_1$ and $y \in S_2$ returns whether $\phi(x) = y$. To aid us, there is a predicate $\mathbb{P} : U \to \{T, \perp\}$ such that $\mathbb{P}(\phi(x)) = \phi(x)$ for all $x \in S_1$. Then the density of $\mathbb{P}$ on $A \subseteq U$ is defined to be $|\mathbb{P}^{-1}(\{T\}) \bigcap A|/|A|$. We assume the density of $\mathbb{P}$ on U, S_1 and S_2 is about the same, say d.
According to lemma 27 in [10]: For a fixed success probability of

$$1 - \frac{1}{e}$$

the algorithm performs $O(\sqrt{N/d})$ operations to build the lists of elements that are checked to find a collision from S_1 and S_2. It queries the **FindFunction** at most $d.N$ times, with a cost of $\mathcal{C}_{\mathcal{FF}}$ per query.
The optimal run time complexity for this algorithm is when $d = N^{-1/3}.\mathcal{C}^{-2/3}$, and given as $O(N^{2/3}.\mathcal{C}^{1/3})$. It reduces to $O(N^{2/3})$, if the solver runs in polynomial-time.
when the size of the lists is approximately $\sqrt{dN}$.
The algorithm performs $O(\sqrt{N/d})$ operations to build the lists and queries the solver function at most $d.N$ times with a cost of $\mathcal{C}$ per query.
The optimal run time complexity for this algorithm is when $d = N^{-1/3}.\mathcal{C}^{-2/3}$, and given as $O(N^{2/3}.\mathcal{C}^{1/3})$. It reduces to $O(N^{2/3})$, if **FindFunction** runs in polynomial-time.

The foundation of this algorithm is used in the Map your Neighborhood algorithm to efficiently manage the complexity of searching through large graphs. The probability of finding a potential collision given by the birthday paradox can be used for selecting subsets of nodes from the graph to increase efficiency of finding a potential match.

3 The Map your Neighborhood algorithm

As defined in [2], the Map your Neighborhood algorithm turns the algebraic problem of finding isomorphism between polynomials into a combinatorial task, retrieving partial information from two exponentially large graphs. The primary goal of the algorithm is to find a 'collision' between the vertices of two graphs,

each representing equations within the quadratic map.
If such a collision is found, the bigger goal of finding the transformation matrices $\mathbf{S}$ and $\mathbf{T}$ becomes achievable in polynomial time with the use of algebraic algorithms. This is because the collision provides enough linear equations in the entries of $\mathbf{S}$ and $\mathbf{T}$, so the overall algebraic system that is needed to solve $\mathbf{S}$ and $\mathbf{T}$ becomes easily solvable. This concept is explained in detail in section 2 of [2]. Briefly, Let f, g be polynomial maps, x be any vector in the vector space, α be an arbitrarily chosen vector and β be the image of α under $\mathbf{S}$. If the image of $\mathbf{S}$ is known at one arbitrary point of the vector space, that is, $\beta = \mathbf{S}.\alpha$, then:

$$\forall x.g(x) = \mathbf{T}f(\mathbf{S}x) \Longleftrightarrow \forall x.g(x+\alpha) = \mathbf{T}f(\mathbf{S}x+\beta)$$

Concretely, if two graphs (V_1, E_1) and (V_2, E_2) are isomorphic, and if ϕ is an isomorphism between them, then $u \in V_1$ and $\phi(u) \in V_2$ have the same degree, which means that they have the same number of neighbors.
This algorithm builds on this idea and extends this concept by looking at the whole connected component of the vertices to distinguish between vertices of the same degree.
To increase efficiency, this algorithm targets vertices of a specific degree in order to find the"right pair" that satisfies $x = y \cdot S$, where x and y are elements in $\mathbb{F}_q^n$ corresponding to vertices in the graph. This implies that the connected component of x is isomorphic to the connected component of y.

The following sections explain the different building blocks and working of the Map your Neighborhood algorithm implemented in the thesis.

3.1 Finding neighbors - BuildSubgraph()

This algorithm uses Breadth-First search to find the connected component of the vertex given as input. The depth parameter for the search constrains the search to only consider the neighborhood of radius d of a vertex.
It builds a subgraph of the component by visiting the neighbors of the vertex (elements in the kernel of the differential associated with the vertex) and counting the number of neighbors that it has. It does this by computing the dimension of the kernel of the differential matrix at that neighbor.

To reduce the search space and make the search more accurate, the neighbors with a higher degree, that is with more neighbors are considered first. So, the neighbors are sorted in descending order based on the number of neighbors they have.
This provides information about the connectivity patterns and structural properties of the graph, which helps in identifying similar patterns that could indicate isomorphism between graphs.

3.2 Distinguishing between vertices - HashTable()

The algorithm defined in [2] utilises the Canonical Graph Labelling algorithm, which relabels the vertices of a graph G to produce a graph $Canon(G)$ that is isomorphic to G.
If two graphs G and $\mathcal{H}$ are isomorphic, then $Canon(G)$ equals $Canon(\mathcal{H})$. In the context of disconnected graphs G^*, the connected component of the vertex x can be denoted as $\mathcal{C}_x$, and the relation $x = y \cdot S$ implies that $Canon(\mathcal{C}_x)$ equals $Canon(\mathcal{C}_y)$.
The hash function thus used in the Map your Neighborhood algorithm is $H : u \rightarrow Canon(\mathcal{C}_u)$. And it is hoped that it behaves as a good hash function, meaning there that false positives, which is for pairs (x, y), $H(x) = H(y)$, but $x \neq y.S$ should be rare.
While the algorithm described in this thesis does not use canonical labeling, it leverages the concept to differentiate between vertices.
The hash value of the subgraphs formed by the connected components is computed and then mapped to the corresponding vertex. To facilitate this process, a hash table is used to store the vertices at the index corresponding to the hash value of the subgraphs of the vertices. So the hash value of the subgraphs of the vertices stored at the an index is the same. By going through this hash table, the algorithm can identify vertices with same connected components (i.e. vertices that are stored at same indices).

3.3 Finding a match - FindMatch()

To find an isomorphism between two sets of polynomials, the hash table of both sets is built of size — *limit*. Vertices that will be stored in the hash table are chosen randomly. Then by iterating through the hash tables of both sets, the vertices at the same non-empty indices are examined by the Check-Match() function that is responsible for computing the transformation matrix **S** between $\mathcal{F}$ and $\mathcal{P}$. We suppose that it checks whether $x = yS$ for some S_1 and, if so, returns S. The algorithm behind this function consists of solving an instance of the inhomogenous QMLE problem and is out of scope for this thesis.

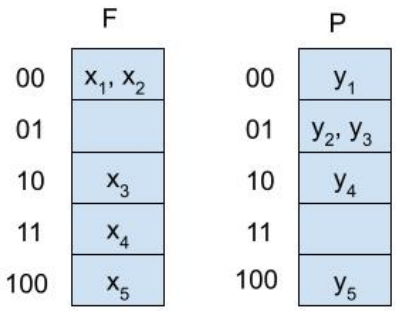
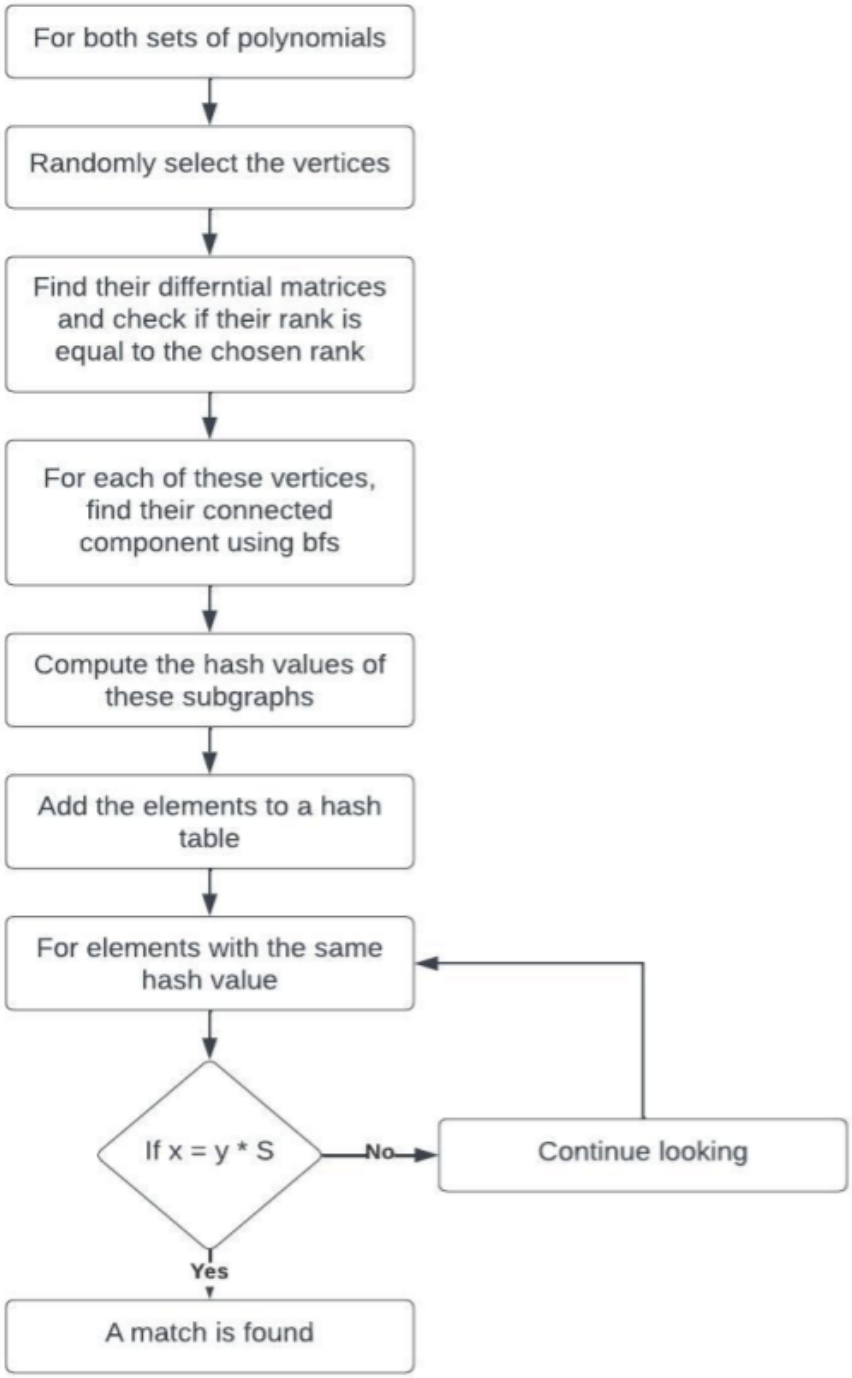


Figure 2

From figure 1, these are $(x_1, y_1), (x_2, y_1), (x_3, y_4), (x_5, y_5)$. This function checks if the vertices satisfy the relation for isomorphism (i.e. the relation $x = y * S$ holds).
If a match is found by this function for any x and y, then S is a solution to the isomorphism problem between the two sets of polynomials. The limit parameter used in the algorithm restricts the number of vertices which are stored in the hash table, so there is a trade-off between the computational efficiency of the algorithm and the likelihood of finding a solution.
Since this is a collision-based algorithm, the probability of finding a solution follows from the birthday paradox and can also provide the basis on which the limit parameter should be selected for the most efficient results.
The following flowchart represents the logic behind algorithm 1:


For both sets of polynomials
Randomly select the vertices
Find their differntial matrices and check if their rank is equal to the chosen rank
For each of these vertices, find their connected component using bfs
Compute the hash values of these subgraphs
Add the elements to a hash table
For elements with the same hash value
If x = y * S
No
Continue looking
Yes
A match is found

3.4 Algorithm 1

See appendix (Section 7) for the code

Algorithm 1 Map your Neighborhood

Function BuildSubgraph($\mathcal{F}, a, depth$)
$subgraph \leftarrow$ breadth-first search starting from a until $depth$
return $subgraph$

Function HashTable($\mathcal{F}, limit, rank, depth$)
$H \leftarrow$ an empty hashtable; $l \leftarrow 0$;
while $l \leq limit$ **do**
    $a \leftarrow$ random element from $\mathbb{F}_q^N$
    if rank of differential matrix of $\mathcal{F}$ at $a = rank$ **then**
        $A \leftarrow$ BuildSubgraph($\mathcal{F}, a, depth$)
        $h_a \leftarrow$ hash(A)
        $H[h_a] \leftarrow a$
    end if
    $l \leftarrow l + 1$
end while
return H

Function FindMatch($\mathcal{F}, \mathcal{P}$)
$X \leftarrow$ HashTable($\mathcal{F}$, limit, rank, depth)
$Y \leftarrow$ HashTable($\mathcal{P}$, limit, rank, depth)
$index \leftarrow 1$
while $index \leq$ limit **do**
    if $X[index] \neq$ None and $Y[index] \neq$ None **then**
        $isMatch, \mathbf{S} \leftarrow$ CheckMatch($X[index], Y[index]$)
        if $isMatch =$ True **then**
            return S
        end if
    end if
    $index \leftarrow index + 1$
end while
return no solution found

3.5 Remarks on Algorithm 1

Establishing the complexity of this algorithm is challenging, as it depends on the structural differences between the graphs formed by the elements in different characteristics of the finite field.
Algorithm 1 is based on the framework of the Map your Neighborhood algorithm given by the authors of [2], and the analysis of Algorithm 1 is based on the work presented by the authors of [2] and [10]. The algorithm is designed independently of the value of q. However, its properties have only been discussed

when $q = 2$ and is inconclusive for other values, but the authors of [2] believe that the complexity and/or success probability degrade exponentially fast when q grows.

In this thesis, the implementation of the Map your Neighborhod algorithm (Algorithm 1) differs from the approach used in Algorithm 3 as outlined by the authors of [2], in certain aspects.
Algorithm 1 deviates from the canonical labeling approach employed in Algorithm 3. While both algorithms use hashing, it is utilized in different ways. In Algorithm 3, the hash function acts as a filter to exclude vertices that do not meet the specific criteria being tested for checking for isomorphism, which involve a Breadth-First Search (BFS) to explore the neighborhood of the vertices. In contrast, in Algorithm 1, hashing is used more directly to test for isomorphism, and the criteria used for filtering vertices is the kernel dimension, determined by the differential operator associated with each vertex.

Similar to Algorithm 3, Algorithm 1 assumes access to a polynomial-time solver for the inhomogenous instance of the QMLE problem, used in the CheckMatch() function. This function is used to find transformation matrices from inhomogeneous instances where a collision is found, reducing the complexity of solving for $\mathbf{S}$ and $\mathbf{T}$[10].
The success probability of the algorithm follows from the principles of the birthday paradox[13], which states that the probability of finding the "right pair" (x, y) such that $x = y * S$ is $1 - 1/e$[2]. The parameters that are significant in determining the time and space complexity of Algorithm 1 are, *depth* and *limit*. The complexity of building the subgraphs and its size formed by the BuildSubgraph() function depends on the extent of the Breadth-First search, determined by the *depth* parameter used. Although the complexity of the HashTable() function and the FindMatch() function depend on the value of the parameter *limit* used, setting the number of entries managed and comparisons made, respectively. These parameters are critical in defining both the time and the space complexity of the algorithm.
The following section presents the results obtained through the experimental evaluation of the algorithm.

4 Experimental results

The algorithm was implemented using a software called SageMath[1], as it provides built-in tools for matrix generation, rank calculation, run time calculations, etc. This section outlines the experimental setup used to evaluate the performance of the Map your Neighborhood algorithm for the odd characteristic of the finite field.
This setup presents the various configurations tested, including variations in matrix size, rank, depth, and the limit parameter. More specifically:

- **Matrix size**: Size of the matrices representing the polynomial sets being tested for isomorphism (size of $\mathcal{F}$ = size of $\mathcal{P}$), affecting the overall computational complexity.
- **Finite field**: Size of the finite field, focusing on $q = 2$ and $q = 3$.
- **Rank**: Rank of the differential matrix, chosen as the criteria to test the vertices for potential isomorphism.
- **Depth**: The *depth* parameter for building subgraphs in the BuildSubgraph() function
- **Limit**: The *limit* parameter used in the HashTable() and FindMatch() function.
- **Avg. distinct hash values**: Average of the number of hash values obtained in the hash table divided by the *limit*, over 50 runs.
- **No. of matches found**: Number of times the FindMatch() function returns a solution over 50 runs.
- **Success probability**: Percentage of the number of times a match is found out of 50 runs.

Matrix size	**Finite field (q)**	**Rank**	**depth**	**Limit**	**Avg. distinct hash values**	**No. of matches found**	**Success probability (out of 50 runs)**
8 x 8	2	6	4	100	18	10	20%
8 x 8	3	6	4	100	6	2	4%
8 x 8	5	6	4	100	4	0	0%
8 x 8	2	6	5	100	25	5	10%
8 x 8	3	6	5	100	6	2	4%
8 x 8	2	6	5	200	26	7	14%
8 x 8	3	6	5	200	7	3	6%
12 x 12	2	10	4	300	54	1	2%
12 x 12	3	10	4	300	14	0	0%
12 x 12	2	10	5	500	96	3	6%
12 x 12	3	10	5	500	19	1	2%

4.1 What we test, how and why

4.1.1 What we test

The parameters were varied to show the variations in the algorithm's functionality and to compare its working for the odd and even characteristic of the finite field in each configuration.

Matrix size and rank
Values: 8 x 8 and 12 x 12 size matrices.
Corresponding ranks of 6 and 10, respectively.
Purpose: To test the algorithm under varying levels of complexity. The matrix size impacts the computational load of the algorithm as it determines the number of equations that have to be handled. Larger matrix sizes led to very long run times as the implementation was primarily for testing and not optimized for handling such complexity.
The rank determines the complexity of finding a solution within the matrices. These specific ranks were selected through a trial and error process, ensuring an adequate solution space for effective comparison across both characteristics of the finite field.

Finite Field Characteristic (q)
Values: $q = 2$ and $q = 3$.
Purpose: To study the algorithm's performance in the odd characteristic of the finite field. Since the behavior of the algorithm has previously been studied for $(q = 2)$, it is used as a baseline for the performance of the algorithm. The behavior of the algorithm for $(q = 3)$ is determined through a comparative analysis. Due to computational constraints, larger values of q could not be considered as it yielded indefinitive results such as in the case when $q = 5$.

Depth and Limit
Values: 4 and 5, and ranging from 100 to 500, respectively.
Purpose: The depth parameter influences the number of neighbors explored for building the subgraph, potentially uncovering more detailed structures at higher depths. The limit parameter determines the number of candidates tested for a potential isomorphism. Thus, the time and memory complexity of this algorithm is directly dependent on these parameters, and are chosen for optimizing computational complexity and success probability.

4.1.2 How we test

To study the behavior of the algorithm, a comparative analysis is used by collecting and analysing data such as the average number of distinct hash values generated, the total number of matches found, and the success probability over 50 runs. The following methods are used to ensure validity of the analysis:

Consistent Configurations: All other experimental conditions (such as matrix size, rank, depth, and limit) are kept constant across tests in both characteristics of the finite field. This ensures that the observed differences in performance can be attributed to the field characteristic.

Simultaneous testing: Tests for both $q = 2$ and $q = 3$ are conducted in parallel to minimize external variations that may affect the outcome, such as variations in the computational environment.

Repetition: The tests were repeated 50 times for each set of parameters, in both characteristics of the finite field. This provides a data for comparison of the observed results and decreases inaccuracy in the results caused by outliers.

The data that is used for analysis:

Avg. distinct hash values: This is used to analyse the impact of the increased complexity and number of elements on the graph traversal, introduced by larger values of the finite field, $q = 3$ in this case. It helps in assessing whether the hash function is discriminating enough in denser structures.

No. of Matches Found and Success Probability: This data reflect the al-

gorithm's effectiveness in identifying potential isomorphic matches. The success probability provides a direct measure of the algorithm's efficiency and behavior for the odd characteristic of the finite field as compared to the even characteristic of the finite field.

4.1.3 Why we test

The experimental evaluation of the algorithm is important for various reasons:

Verifying Theoretical Predictions: The properties of the Map your Neighborhood algorithm have been studied for the case when $q = 2$, and the authors of [2] believe that the complexity and/or success probability of this algorithm degrades exponentially fast when q grows. But they do not have a definitive conclusion for it. The experiment helps in empirically verifying if the practical behavior of the algorithm aligns with the theoretical hypothesis.

Optimization of the algorithm: Identifying the inefficient areas in the performance of the algorithm for $q = 3$, provides insight for future optimizations, increasing the scope of application of the algorithm.

Cryptographic Significance: In multivariate cryptography, the security of cryptographic systems relies on the difficulty of solving the IP problem. Understanding the algorithm's effectiveness for different finite field characteristics is important to determine the values that need to be selected in order to ensure security in cryptographic systems.

4.2 Interpretation of the results

The main observations that can be made from the experimental results for the algorithm in the odd characteristic of the finite field:

1. The success probability of the algorithm decreases with an increase in the size of the matrix, for all sets of parameters
2. The hash values of the subgraphs formed by the Breadth-First search are not very distinct on average and do not increase significantly with more depth.
3. The success probability is relatively higher for a bigger value of the Limit: the success probability increases from 4% to 6% when the limit increases from 100 to 200 in the case when $q = 3$.

These observations are not conclusive as they can be attributed to computational and implementation inefficiency such as non recursive function definitions decreasing the efficiency. Thus, they are interpreted by alluding to the the results presented by the authors of [2] in Table 2, about their implementation of the Map your Neighborhood algorithm, shown in Figure 3:

n	q	generating U and V	finding collisions	$\|U\|$	$\mathcal{N}$
16	2	3.6 s	1s	64	6
24	2	123 s	13s	836	5
32	2	61 min	200s	11585	2
40	2	31 h	2h	165794	7

Table 2. Experimental results on Algorithm 3

Figure 3

According to the results in Table 2, it can be concluded that the time taken for generating the hash tables and finding a collision as well as the size of the hash table increases with the matrix size.
This also implies that for larger matrices, more number of vertices need to be checked to find a collision which aligns with the principles of the birthday paradox. Observation 3 proves this further as the Limit parameter determines the number of vertices checked.

Observation 2 indicates that the traversal method or the hash function is not able to accurately discriminate between the vertices, potentially due to the increased number of elements and denser structures. A smaller number of distinct hash values also indicates the occurrence of more false collisions, causing the CheckMatch() function to fail to find a solution frequently, reducing efficiency and accuracy of the algorithm.

Observations about the algorithm's behavior through a comparative analysis:

1. The success probability of the algorithm is consistently higher when $q = 2$ as compared to $q = 3$.

2. The average number of distinct hash values are consistently lower for $q = 3$ than $q = 2$. This difference can be seen more significantly as the size of the matrix and depth increase.

3. The average distinct hash values found in $q = 5$ is lesser than in $q = 3$ for the same set of parameters.

These observations can be interpreted using the difference in the theory of the finite fields when $q = 2$ as compared to $q = 3$.
The proportion of nodes that have connected components of size $q - 1$, grows like $1 - 1/q^2$ as shown by lemma 2 in [2]. This leads to a richer structure of the disconnected linearity graph with more isolated small connected components when $q = 2$[2]. As q increases, the connected components become larger and less isolated. This can also potentially lead to more variety in the connected components, that cannot be accurately captured by the hash function making it less discriminating.

For $q = 2$, the connected components of the vertices form triangular structures due to the algebraic properties of the elements in the field. The connected components when $q \neq 2$ form random trees[2].
This is a potential reason behind observations 2 and 3. Due to the increased complexity in traversal and the potential occurrence of more similarities in the connected components, the hash function becomes less discriminating for $q = 3$ than $q = 2$. This is further seen in observation 3 as the hash values become even less distinct as q increases. All of this contributes to the occurrence of observation 1.

It can also be observed that the behavior of the algorithm aligns with the theoretical predictions in the case when $q = 2$. The hash values become more distinct for a larger depth parameter and size of the matrix, indicating that a larger section of the connected components is explored, revealing more uniqueness. Increasing the limit parameter leads to an increase in the success probability when $q = 2$, relative to the other parameters. Another observation that can be made is that a smaller value of the limit parameter for matrix of size 8x8, yields a higher success probability than for matrix of size 12x12. Suggesting that the smaller matrix size increases the likelihood of collision detection with fewer elements as opposed to a larger set, following the principles of the birthday paradox.

5 Conclusion

The primary objective of this thesis was to explore the behavior of the Map your Neighborhood algorithm for the finite field of odd characteristic specifically $q = 3$. This was achieved through a comparative study between the performance of the algorithm in $q = 2$ and $q = 3$. The existing work done on this algorithm focused on the case when $q = 2$ in [2], is used as the foundation of this thesis and is instrumental in understanding the theoretical aspects of the working of the algorithm.
The experimental data provided a basis for assessing the algorithm's performance across the finite field of odd and even characteristics. The results illustrated that the algorithm was less effective for $q = 3$ than $q = 2$. The theory of the differences in quadratic forms in finite fields of odd characteristics and characteristic two was used to validate the experimental results, stipulating that the added complexity for larger values of q impacts the algorithm's ability to accurately distinguish between connected components of the vertices, leading to more inefficiency and inaccuracy in finding isomorphisms.
Additionally, the study provided insights that help in understanding the reliability and scalability required, to employ the algorithm in testing the security of cryptographic schemes based on the QMLE problem in finite fields of odd characteristic.
In conclusion, the Map your Neighborhood algorithm was less effective in the finite field of odd characteristic, with a success probability for $q = 3$ less

than/equal to half of the success probability for $q = 2$ in all settings. Further research and an in-depth complexity analysis of the algorithm based on the properties of the finite fields of odd characteristics, is necessary to understand its overall effectiveness and limitations for cryptanalysis.

6 Future work

Currently, the complexity of the best algorithm for solving QMLE for a general q is $O(q^{2N/3})$.
The Map your Neighborhood has a complexity of $O(q^{n/2})$ when $q = 2$. Further work is needed to find the complexity of this algorithm for $q > 2$. And in determining definitively if this algorithm is faster than the current best algorithm for a general q.
Work related to optimizing the algorithm to better handle fields with odd characteristics, potentially through adjustments in the hashing mechanism or traversal algorithm, can help in adapting the algorithm for larger values of q. Testing the algorithm across a wider range of odd characteristics would provide a more generalized understanding of its behavior and limitations.
Theoretical work on quadratic forms in the odd characteristic is needed to conclusively identify how the performance of the algorithm is affected, and develop methodologies to improve performance for a general q. Furthermore, this will give insight into the parameters that can increase the security of cryptographic systems that rely on the hardness of solving the QMLE problem.

References

[1] URL: https://www.sagemath.org/.

[2] Fouque PA. Bouillaguet C. and Veber A. *Graph-Theoretic Algorithms for the Isomorphism of Polynomials Problem.* Johansson, T., Nguyen, P.Q. (eds) Advances in Cryptology – EUROCRYPT 2013. EUROCRYPT 2013. Lecture Notes in Computer Science, vol 7881. Springer, Berlin, Heidelberg. 2013. URL: https://eprint.iacr.org/2012/607.pdf.

[3] Wikipedia contributors. *Examples of vector spaces.* Wikipedia, The Free Encyclopedia. 2023. URL: https://en.wikipedia.org/w/index.php?title=Examples_of_vector_spaces&oldid=1187701944.

[4] Wikipedia contributors. Wikipedia, The Free Encyclopedia. 2024. URL: https://en.wikipedia.org/w/index.php?title=Affine_group&oldid=1221937402.

[5] Jintai Ding and Bo-Yin Yang. *Multivariate Public Key Cryptography.* 2009. URL: https://link.springer.com/content/pdf/10.1007/978-3-540-88702-7_6.pdf.

[6] Shimon Even and Yishay Mansour. *A Construction of a Cipher from a Single Pseudorandom Permutation.* J. Cryptology. 1997.

[7] Jean-Charles Faugere and Ludovic Perret. *Polynomial Equivalence Problems: Algorithmic and Theoretical Aspect.* Vaudenay, S. (ed.) EUROCRYPT '06. Lecture Notes in Computer Science, vol. 4004, pp. 30–47. Springer. 2006.

[8] Patarin Jacques. *Hidden Fields Equations (HFE) and Isomorphisms of Polynomials (IP): Two New Families of Asymmetric Algorithms.* Maurer U. (eds) Advances in Cryptology — EUROCRYPT '96. EUROCRYPT 1996. Lecture Notes in Computer Science, vol 1070. Springer, Berlin, Heidelberg. 1996. URL: https://link.springer.com/chapter/10.1007/3-540-68339-9_4.

[9] Louis Goubin Nicolas Courtois Jacques Patarin. *Improved algorithms for isomorphisms of polynomials.* Nyberg, K. (eds) Advances in Cryptology — EUROCRYPT'98. EUROCRYPT 1998. Lecture Notes in Computer Science, vol 1403. Springer, Berlin, Heidelberg. 1998. URL: https://doi.org/10.1007/BFb0054126.

[10] Simona Samardjiska Krijn Reijnders and Monika Trimoska. *Hardness estimates of the Code Equivalence Problem in the Rank Metric.* Cryptology ePrint Archive. 2021. URL: https://eprint.iacr.org/2022/276.

[11] Imai H. Matsumoto T. *Public Quadratic Polynomial-Tuples for Efficient Signature-Verification and Message-Encryption.* In: Barstow, D., et al. Advances in Cryptology — EUROCRYPT '88. EUROCRYPT 1988. Lecture Notes in Computer Science, vol 330. Springer, Berlin, Heidelberg. 1998. URL: https://doi.org/10.1007/3-540-45961-8_39.

[12] J. Patarin. *The Oil and Vinegar Signature Scheme*. presented at the Dagstuhl Workshop on Cryptography. september 1997 (transparencies).

[13] S. Vaudenay. *A Classical Introduction to Cryptography: Applications for Communications Security*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. 2005.

7 Appendix

```
from sage.matrix.all import *
from collections import deque
import random

def Encrypt(f, x):
    """
    Evaluates polynomial f in x.
    """
    return vector([x.dot_product(x*M) for M in f])

def ChangeCoordinatesLeft(a, T):
    """
    Computes T o a.
    """
    Ta = [zero_matrix(T.base_ring(), N, N) for _ in range(k)]
    for i in range(len(a)):
        for j in range(k):
            Ta[i] += T[j, i] * a[j]
    return Ta

def ChangeCoordinatesRight(a, S):
    """
    Computes a o S.
    """
    return [S * M * S.transpose() for M in a]

def ReducedForm(a):
    """
    Puts the matrices in upper-triangular form.
    """
    a_r = a
    for i in range(len(a)):
        for j in range(N):
            for k in range(j + 1, N):
                a_r[i][j, k] += a[i][k, j]
                a_r[i][k, j] = 0
    return a_r
```

```
def ProblemGen():
    """
    Generates a quadratic homogeneous instance of QMLE.
    """
    while True:
        S = random_matrix(GF(q), N, N)
        T = random_matrix(GF(q), k, k)
        try:
            S_inv = S.inverse()
            T_inv = T.inverse()
            break
        except ZeroDivisionError:
            continue
    F = [random_matrix(GF(q), N, N) for _ in range(k)]
    P = ChangeCoordinatesLeft(ChangeCoordinatesRight(F, S), T_inv)
    return S, T, F, P

def Diff(f, x, y):
    """
    Computes the differential in one point: f(x+y) - f(x) - f(y) + f(0).
    """
    return Encrypt(f, x + y) - Encrypt(f, y) - Encrypt(f, x) + Encrypt(f, zero_vector(K, N))

def DMat(PK, y):
    """
    Returns the matrix that describes the function: x -> f(x+y) - f(x) - f(y) + f(0).
    """
    b = V.basis()
    M = matrix([Diff(PK, y, b[i]) for i in range(N)])
    return M

def DMat2(basis, z):
    """
    Uses a precomputed basis of the differential to compute the differential.
    """
    D = zero_matrix(K, N, k)
    for j in range(N):
        D += z[j] * basis[j]
    return D

def neighbors_concatenate(arr1, arr2):
    for i in arr2:
        arr1.append(i)
    return arr1
```

```
def find_neighbors_sorted(x, F_basis):
    M = DMat2(F_basis, x)
    kernel = M.kernel()
    neighbors_vector = []
    neighbors_nb_neighbors = []
    for neighbor in kernel:
        if neighbor == V.zero_vector():
            continue
        M = DMat2(F_basis, neighbor)
        dim = M.kernel().dimension()
        if dim == N:
            continue
        neighbors_vector.append(neighbor)
        neighbors_nb_neighbors.append(dim)
    #sort neighbors by the number of neighbours that they have
    i = 0
    while i < len(neighbors_nb_neighbors) - 1:
        max_val = neighbors_nb_neighbors[i]
        max_index = i
        j = i + 1
        while j < len(neighbors_nb_neighbors):
            if neighbors_nb_neighbors[j] > max_val:
                max_val = neighbors_nb_neighbors[j]
                max_index = j
            j = j + 1
        #swap
        temp = neighbors_nb_neighbors[i]
        neighbors_nb_neighbors[i] = neighbors_nb_neighbors[max_index]
        neighbors_nb_neighbors[max_index] = temp
        temp = neighbors_vector[i]
        neighbors_vector[i] = neighbors_vector[max_index]
        neighbors_vector[max_index] = temp

        i = i + 1
    return neighbors_vector, neighbors_nb_neighbors

#it is not recursive
def find_subgraph(x, F_basis, max_depth):
    M = DMat2(F_basis, x)
    kernel = M.kernel()
    neighbors_vector = []
    neighbors_vector.append(x)
    neighbors_nb_neighbors = [kernel.dimension()]
    cursor = 0
    depth = 1
    while depth < max_depth:
```

```
        new_neighbours_vector = []
        new_neighbours_nb_neighbors = []
        while cursor < len(neighbors_vector):
            neighbors_vector_temp, neighbors_nb_neighbors_temp = find_neighbors_sorted(neigh
            neighbors_concatenate(new_neighbours_vector,neighbors_vector_temp)
            neighbors_concatenate(new_neighbours_nb_neighbors,neighbors_nb_neighbors_temp)
            cursor = cursor + 1
        neighbors_concatenate(neighbors_vector, new_neighbours_vector)
        neighbors_concatenate(neighbors_nb_neighbors, new_neighbours_nb_neighbors)
        depth = depth + 1
    return (neighbors_nb_neighbors)


def build_list(A, limit, depth):
    arr = [None] * limit
    size = 0
    rank = 5
    while size < limit:
        xx = V.random_element()  # Generate a random element from V
        dmat = DMat(A, xx)
        rank_x = dmat.rank()
        if rank_x == rank:
            x_tuple = tuple(find_subgraph(xx, A, depth))
            ind = hash(x_tuple) %limit
            arr[ind] = xx
            size += 1
    return arr
#values chosen for the experiment is written in brackets
q = #finite field (2,3,5)
N = #size of the matrix (8,12)
k = #no. of polynomials in the map (8,12)
nb_tests = #no. of runs (50)

K = GF(q) #Finite field over q
V = VectorSpace(K, N)
basis_V = basis(V)

print(f"N = {N}, k = {k}, K = GF({q})")

for i in range(nb_tests):
    S, T, F, P = ProblemGen()
    DMat_basis_F = [DMat(F, basis_V[i]) for i in range(N)]
    DMat_basis_P = [DMat(P, basis_V[i]) for i in range(N)]

    num_V = len(V.list())
    i = 0
```

```
depth = #depth for BFS search (4,5)
for yy in random.sample(V.list(), num_V):
    if yy == V.zero_vector():
        continue
    xx = yy*S

    # Compute the differential matrix M
    M_F = DMat2(DMat_basis_F, xx)
    M_P = DMat2(DMat_basis_P, yy)

    # Compute the rank of M
    rank_F = M_F.rank()

    if rank_F == #chosen rank (6,10):
        print("Rank of M (F) =", rank_F)
        print(find_subgraph(xx, DMat_basis_F, depth))
        print(find_subgraph(yy, DMat_basis_P, depth))
        break

found = False
limit = #size of lists used for finding collision (100,200,300,500)
arr_F = build_list(F, limit,depth)
arr_P = build_list(P, limit, depth)
i = 0
while i < limit and not found:
    for ind in range(limit):
        xx = arr_F[ind]
        yy = arr_P[ind]
        if xx is not None and yy is not None:
            if xx == yy *S:
                found = True
                print("matched")
                print(xx)
                print(yy)
                break

    i += 1
if not found:
    print("No match found")
```