Multivariate cryptography and algebraic techniques used in cryptanalysis

Dalvir Singh Mandara

1423101

BSc Mathematics and Computer Science

School of Computer Science, University of Birmingham

Supervisor: Dr. Christophe Petit

April 2018

Abstract. Multivariate cryptographic schemes such as HFE (Hidden field equations) [1] and balanced oil vinegar [2] have been broken, but there are also many schemes that are yet to be broken and are being considered as options for post quantum security, so research in this area is important. We describe the details of both multivariate encryption schemes and multivariate signature schemes as well as a detailed overview of the algebraic techniques used in cryptanalysis and some of their future applications. A common issue with multivariate schemes is the potential for big public keys, in this paper we look at the normal UOV scheme [3] and compare it to a new UOV scheme [4] that reduces the size of the public keys by constructing the keys over a smaller field where less bits are used to represent each coefficient [4]. We illustrate this with implementations of each scheme in SageMath as well as comparisons of the algorithms that are used to generate the keys and signatures in each of the schemes.

Keywords: Multivariate Cryptography, Encryption, Signature Scheme, Field Lifting, Cryptanalysis

All software for this project can be found at: https://git-teaching.cs.bham.ac.uk/mod-ug-proj-2017/dsm501

Contents

1 Introduction

The goal of this report is to provide an extensive account of the inner workings of multivariate cryptosystems and contribute to their understanding as well as a detailed insight of how different algebraic techniques are used in cryptanalysis. In addition, we give the reader a comparison of current and past multivariate schemes and what future work could be expected to achieve. Number theory underpins many of the public key cryptosystems currently used for example RSA (Rivest–Shamir–Adleman) [5], which is dependent on the hardness of factoring products of large primes. [5]. Public key cryptographic schemes are asymmetric systems in which a key that allows for encryption is made public, but decryption occurs using a separate private key that the writer of the scheme knows. Advances in quantum computing and the potential for quantum computers to be made more widely available in the near future raises concerns for the cryptosystems that are being used today. Take RSA [5] as an example, quantum computers would be able to defeat this scheme in polynomial time using Shor's algorithm [6], this is a lot faster than the equivalent case for classical algorithms, which have an exponential time complexity when factoring large primes. Although this algorithm is not yet practical since there are not currently quantum computers large enough to run it, when quantum computers do become more accessible coupled with the advance of quantum algorithms, almost all of the of the public key cryptosystems that are presently being used would be defeated.

Research into schemes that could provide post quantum security has come as a result of this threat, multivariate cryptographic schemes are one of the ones at the forefront of this research, it is considered to be a serious option for post quantum security. Such schemes are centralised around the hardness of solving multivariate systems of polynomial equations. Solving systems of multivariate polynomials over a finite field is an NP-complete problem [7] (it cannot be solved in polynomial time). There are no current quantum algorithms that can solve this problem but there are algorithms such as Grover's algorithm [8] that can make certain parts of the brute force approach a lot quicker. Multivariate cryptosystems can be very fast to use, however an overhead is that the public keys of these schemes can get very large. In this paper we have looked at the current UOV (Unbalanced Oil and Vinegar) [3] scheme that remains unbroken and have implemented it in sage, along with a comparison against a new scheme that reduces the key sizes [4] that is also based on UOV. We have implemented both schemes in the hopes of giving the reader an insight into how the new scheme differs to the normal UOV scheme and how it achieves smaller public keys in practice.

We give a detailed account of how multivariate schemes are structured in section 2, we present the differences between encryption schemes and signature schemes including UOV [3] and its algorithms in sections 3 and 4. Section 5 goes into detail about the methodology behind different algebraic cryptanalysis techniques including the Kipnis-Shamir cryptanalysis of balanced oil vinegar [9] and

Grobner basis calculations. Finally in section 6 we present our implementation of the standard UOV scheme and the new lifted UOV scheme before discussing our outcomes in section 7 and concluding in section 8.

2 Multivariate Cryptography

Multivariate public key cryptosystems (MPKC) are cryptosystems whose public keys are sets of multivariate polynomial equations. These equations are usually quadratic for efficiency purposes which will be explained later. The construction of the the public key requires an easily invertible core map $F : \mathbb{F}^n \to \mathbb{F}^m$, two invertible linear maps $S : \mathbb{F}^m \to \mathbb{F}^m$ and $T : \mathbb{F}^n \to \mathbb{F}^n$. The public key is then given by the composition of these three maps, with the core map F being in the middle, i.e $P = S \circ F \circ T$. The private key which allows for the inversion of the public key is given by (S, F, T). The set of equations that form the public key, P, looks as follows: $P(x_1, ..., x_n) = (p_1(x_1, ..., x_n), ..., p_m(x_1, ..., x_n))$. Where m and n are integers and $p_i(x_1, ..., x_n)$ are all multivariate polynomials over a finite field. The strength of MPKC relies on the hardness of solving such systems of multivariate polynomial equations which is known to be an NP-Complete problem [7].

2.1 Size of the public key

For a system of non linear polynomial equations we will let d = degree of the polynomials in the system, m = the number of equations in the system, n = the number of variables each polynomial is made of. The size of the public key P is given by the following equation $Size_P = m \cdot E$ where E = the number of terms in each polynomial of degree $\leq d$. Making unordered choices using combinatorics we obtain that, $\binom{n+d-1}{d}$ = the number of monomials of degree d and from this we get that $\binom{n+d}{d}$ = the number of monomials of degree $\leq d$, which is E. Putting all this together we see that the size of the public key is $Size_P = m \cdot \binom{n+d}{d} \sim O(n^{d+1})$ for $m \approx n$. From this it is clear that for $d > 2$ the public key size gets very large very quickly which is a drawback. For this reason along with maintaining efficiency most MPKCs use $d = 2$, forming quadratic polynomials. This leads onto the MQ problem which will now be stated formally.

2.2 The MQ problem

We give the MQ problem below as defined in Ward Beullens and Bart Preneels paper describing the new scheme [4].

MQ Problem. Given a quadratic polynomial map $P : \mathbb{F}_q^n \to \mathbb{F}_q^m$ over a finite field $\mathbb{F}_q$, find $\mathbf{x} \in \mathbb{F}_q^n$ that satisfies $P(\mathbf{x}) = 0$ [4].

The MQ problem is NP-complete [7] which means there is no known algorithm that can solve the problem in polynomial time and it is thought to be hard on average for classical and quantum computation (in contrast to factoring that is used in RSA today). Quadratic constructions of the public key are mostly used for efficiency reasons as stated above but also for structural consideration. Since any system of higher degree polynomial equations can essentially be reduced to a system of quadratic equations which implies that if you are able to solve a quadratic equation then you can solve any of the other higher order equations. It is important to note that this is one of the reasons why the security of MPKCs is based on the MQ problem, furthermore, every cryptosystem can be described as a system of multivariate nonlinear polynomial equations which makes the MQ problem a principal problem in cryptography as a whole. This means that if someone where to create an algorithm with the capability of solving the MQ problem in polynomial time, all other cryptosystems would be threatened. The security of MPKC's is also reliant on a problem concerning the isomorphisms of polynomials. This is introduced by first giving a definition.

2.3 The isomorphism of polynomials

Definition. Two polynomial systems $A : \mathbb{F}^n \to \mathbb{F}^m$ and $Z : \mathbb{F}^n \to \mathbb{F}^m$ are isomorphic if there exists linear affine maps N_1, N_2 such that $Z = N_1 \circ A \circ N_2$.

In the case of MPKC's we have $P = S \circ F \circ T$. Applying the definition above it can be seen that P and F are indeed isomorphic. Due to the construction of MPKC's the security of such systems must take into consideration this notion of isomorphism of polynomial systems. For example, given a public key P it is reasonable for an attacker to search for two affine maps S^* and T^* and a quadratic map F^* such that $P = S^* \circ F^* \circ T^*$. Applying the definition above again, implies that P is isomorphic to F^*. This process is known as finding an equivalent public key in terms of an equivalent core map F^* and equivalent affine transformations S^* and T^*. The difficulty of finding an equivalent public key is reliant on the structuring of the core map. However despite this, little is actually known about the complexity of this problem [10] and this is one of the reasons why evaluation of the security of MPKC's is a taxing task. Note that in this problem it is required that the two transformations are bijective, we point out that the equivalent problem without this constraint is called the morphism of polynomials problem which is NP-complete [10].

3 Encryption schemes.

For encryption schemes it is required that $m > n$ to ensure the injectivity of the core map F which means decryption can occur distinctively, where m is again the number of equations in the system and n is the number of variables

in each polynomial. Encryption is done by simply finding the value of the public key polynomial system for a given plain-text, which is a vector over the finite field. For example, consider the public key P given by $P(x_1, \ldots, x_n) = (p_1(x_1, \ldots, x_n), \ldots, p_m(x_1, \ldots, x_n))$ and let the vector $A = (u_1, \ldots, u_n)$ be the plain-text of a message to be encrypted. Computing $P(A) = P(u_1, \ldots, u_n) = (v_1, \ldots, v_n)$ gives us the cipher-text, $(v_1, \ldots, v_n)$. The plain-text message A has been encrypted. This is further illustrated by a simple example given below.

Suppose the public key P is given as follows:

$$\begin{aligned} P = p_0(x_1, x_2, x_3) &= 1 + x_1 + 3 \cdot x_2 + x_3^2 \\ p_1(x_1, x_2, x_3) &= 1 + x_1 + 4 \cdot x_2 + x_3^2 \\ P_2(x_1, x_2, x_3) &= 1 + x_1 + 5 \cdot x_2 + x_3^2 \end{aligned}$$

Now suppose the plain-text $(x_1, x_2, x_3) = (1, 2, 3)$ is to be encrypted. Plugging these values for x_1, x_2 and x_3 into p_0, p_1, and p_2 that form the public key, obtains the cipher-text.

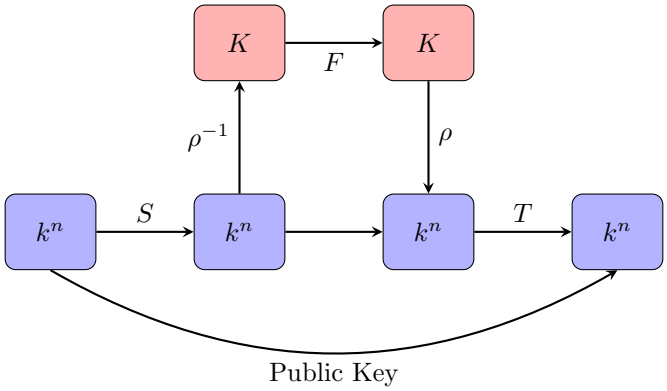
$$p_0(1, 2, 3) = 17 \tag{1}$$
$$p_1(1, 2, 3) = 19 \tag{2}$$
$$p_2(1, 2, 3) = 21 \tag{3}$$

Giving the cipher-text $(v_1, v_2, v_3) = (17, 19, 21)$

To decrypt the cipher-text (v_1, v_2, v_3) the map P needs to be inverted. To do so the private key is needed, once the map P has been inverted the plain-text can be obtained as follows: $A = (u_1, \ldots, u_n) = P^{-1}(v_1, \ldots, v_n)$. An important point to note here is that since the map P is not always necessarily bijective it is appropriate to express the inverse of P as determining the pre-image as opposed to the usual mathematical definition of inverting a function. However, as stated above for multivariate encryption schemes we require that $m > n$ to ensure the injectivity of the core map F, this is required to reduce the number of possible plain-texts so that an encrypted message maps to at most one plain-text. This condition however means that such encryption schemes where there are more equations than variables are more susceptible to attacks such as the direct attack for example. For this reason it is hard to make multivariate encryption schemes secure and therefore multivariate cryptography is not always a very good candidate for encryption schemes. It is however very useful for creating efficient signature schemes which will be discussed later.

3.1 The HFE scheme

The Hidden field equations scheme (HFE) is a multivariate encryption scheme that was first proposed by Patarin in 1996 [1]. Fix a finite field k of size q and a positive integer n. We then choose a degree n extension field K and a map $\rho : K \to k^n$ which is given by $\rho(v_1+v_2y+v_3y^2+\cdots+v_ny^{n-1}) = (v_1, v_2, v_3, \ldots, v_n)$. Form a polynomial $F : K \to K$ such that the hamming weight of F is 2 i.e $F(x) = \sum_{0\leq j\leq i}^{n-1} a_{ij}X^{q^i+q^j} + \sum_{i=o}^{n-1} b_{ij}X^{q^i} + c$, where the coefficients are randomly chosen in K. It is important to note that the degree of this polynomial needs to be controlled as during decryption this equation needs to be solved and if the degree is too high this will cause problems.. The degree of the polynomial will be at most a fixed integer D. Two invertible affine transformations S and T are randomly chosen over k^n. Below is a diagram of how the HFE scheme looks.



The two maps ρ and ρ^{-1} help to move the map from the large field K to the small field k. The public key is made over the small field k and is the trapdoor function given by $P(x_1, \ldots, x_n) = T \circ \rho \circ F \circ \rho^{-1} \circ S(x_1, \ldots, x_n) = p_i(x_1, \ldots, x_n)$ for $i = 1, \ldots n$. From the diagram it can be seen that first the map F is built over the large field K, then the map ρ is used to pull F from the large field to the small field so F becomes a map from $k^n \to k^n$ and then S and T are composed at either end of F. The purpose of S and T is to mix up the maps so that an attacker does not know the private key. The private key is given by (F, S, T). As stated earlier the degree D of the map F needs to be controlled and cannot be too high to allow for decryption as $F(x) = y$ for some y will need to be solved during the decryption process using Berlekamp's Algorithm [11], the complexity of this algorithm is heavily influenced by the degree D. This means if the degree of F is chosen to be too high then then we cannot decrypt. The necessity for D to be controlled is problematic for HFE because it presents exposures to direct attacks and structural attacks. Challenge 1 [1] by Patarin involved a HFE scheme with the following parameters, $q = 2, n = 80$ and $D = 96$ this meant solving a system of 80 equations in 80 variables but $D = 96$ is kept relatively low. Consequently in 2003 Faugere and Joux successfully solved the challenge [12] by computing the corresponding Grobner basis

using the $F4$ algorithm [13]. The weakness with HFE lies in the construction of F, if the degree is low, algebra can used to solve the system as Faugere and Joux showed with their attack, but if the degree is too high then we cannot decrypt.

3.2 The ZHFE scheme

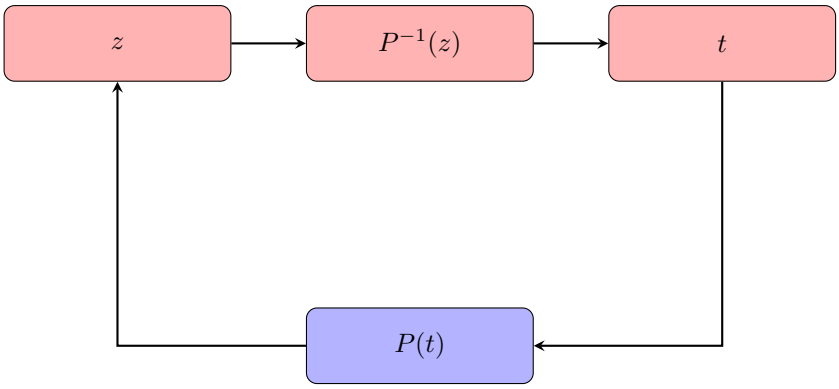
An alternative scheme published by Jintai Ding, John Beana and Jaibarth Porras known as the **ZHFE scheme** [14] claims to avoid this problem of having to keep the degree of the core map F controlled. The ZHFE adapts a fundamental change in the map, a map is build from $k^n \to k^{2n}$ instead of from $k^n \to k^n$ in the original HFE scheme. This is going from a nearly bijective map, to an injective map since we are taking a small space (k) and mapping it to a larger space (k^{2n}) insuring the injectivity of the map and thus allowing decryption to occur uniquely. Bijective maps are problematic because they are easily distinguishable since the kernel of their differentials will always have dimension 0. To construct such a scheme, 2 HFE core polynomials are used over the large field, $F(x)$ and $\tilde{F}(x)$, this takes the map to $k^n \to k^{2n}$. So $H(x) = (F(x), \tilde{F}(x))$ is the core map for the trapdoor function of the ZHFE scheme. Since we have two polynomials of high degree, how do we decrypt? By computing the Frobenius maps of F and $\tilde{F}$ and finding a polynomial Ω of weight 3 [14], such that the $\deg(\Omega) \leq d$ for some fixed small integer d instead of a weight 2 used for the corresponding polynomial in the original HFE scheme. This allows Berlekamp's algorithm [11] to be applied and hence allows for $H(x)$ to be inverted. It is necessary to point out that during decryption solutions are looked for in the finite field itself and not in the extension field which allows us to use the properties of the finite field during calculations.

4 Signature schemes

In contrast to encryption schemes signature schemes require an initial condition that $m \leq n$ where m is again the number of equations in the system and n is the number of variables in each polynomial. This is to ensure the surjectivity of the core map, which means every document or message to be signed has a signature, surjectivity does however mean that there may be more than one valid signature for a given document which is not a problem since we clearly we only need one. Signature schemes have two fundamental steps involved, signature generation and signature verification.

Signature generation: Given a document (or message) q to sign. First use a hash function given by $H : \{0,1\} \to \mathbb{F}^m$ to calculate the hash value of the of the document to be signed, given by $z = H(q)$. Then using the private key which allows us to invert the public map, P. Calculate the signature given by, $t = P^{-1}(z)$. The documents hash value has now been signed with signature t.

Signature verification: Given a pair, (t, z) which is the signature and the hash value respectively, of the document that has been signed. The validity of the signature can be checked by computing $z' = P(t)$ and checking that the following equality holds: $P(t) = H(q)$, i.e $z' = z$. If the equality holds the signature is accepted, if it does not hold then it is rejected. Signature generation and signature validation is shown below in a simple diagram to help visualize the processes.



In the above diagram the red section indicates signature generation and the blue section indicates signature validation. Where z, t, P and P^{-1} are all as described above. An important MPKC signature scheme known as the Oil and Vinegar scheme will now be discussed in detail.

4.1 The unbalanced oil and vinegar signature scheme

The original oil and vinegar (OV) scheme was proposed by Patarin in 1997 [2]. It was subsequently broken by Kipnis and Shamir [9] and so a modified version of the scheme based on the MQ problem was proposed called the Unbalanced Oil and Vinegar scheme (UOV) [3]. The term balanced and unbalanced refers to the number of so called "oil" and "vinegar" variables in each scheme, with the original balanced OV scheme having the same number of each of these variables and the number of each variable type being different in the UOV scheme. The UOV scheme has the following parameters: A finite field $\mathbb{F}_q$, $o, v \in \mathbb{Z}$, fix $n = o + v$, a core map $F : \mathbb{F}^n \to \mathbb{F}^o$, and let $T : \mathbb{F}^n \to \mathbb{F}^n$ be a randomly chosen affine map. The core map F consists of o oil-vinegar polynomials $f_1, \ldots, f_o$ each of the form: [3]

$$f_k(x_1, \ldots, x_n) = \underbrace{\sum_{i=1}^{v}\sum_{j=1}^{v} \alpha_{kij} \cdot x_i \cdot x_j}_{v \cdot v} + \underbrace{\sum_{i=1}^{v}\sum_{j=v+1}^{n} \beta_{kij} \cdot x_i \cdot x_j}_{v \cdot o} + \underbrace{\sum_{i=1}^{n} \delta_{kij} \cdot x_i}_{linear} + \lambda_k$$

In the above equation $(x_1, \dots x_n) = (x_1, \dots, x_v, x_{v+1}, \dots, x_n)$ where $x_1, \dots, x_v$ are the vinegar variables and $x_{v+1}, \dots, x_n$ are the oil variables, and the constants $\alpha_{kij}, \beta_{kij}$ and δ_{kij} are randomly assigned. We see that we have quadratic terms in $v \cdot v$ (vinegar-vinegar), quadratic terms in $v \cdot o$ (vinegar-oil), linear terms in v and o and a constant term λ. It is important to point out here that there are no $o \cdot o$ (oil-oil) terms in the oil-vinegar polynomials, and this is where the name, Unbalanced Oil and Vinegar comes from, the idea is that the oil and vinegar terms do not mix completely which is illustrated by the missing oil-oil terms. This information is displayed in the table below which shows the type of each term present in the oil-vinegar polynomials.

Quadratic terms	$v \cdot v$, $v \cdot o$
Linear terms	v, o
Constant terms	λ

The public key $P : \mathbb{F}^n \to \mathbb{F}^o$ is the composition of the core map $F : \mathbb{F}^n \to \mathbb{F}^o$ and the random linear affine map $T : \mathbb{F}^n \to \mathbb{F}^n$, such that $P = F \circ T$. The private key is the double (F, T) which allows for the inversion of the public key $P = F \circ T$ (for signature generation). It is natural to ask what the purpose of the transformation T is? The core map F has a distinct oil-vinegar shape and the purpose of T is to do a change of basis by mixing up the variables [15]. This means that you cannot distinguish between the oil and vinegar variables anymore which adds a layer of security. The core map F is chosen to be easily invertible, the process of inverting F will now be outlined in detail.

4.1.1 Inversion of the core map

Inversion of F is needed to calculate the pre-image of the hash value under F which is needed to generate a signature. To invert F random values are assigned to the vinegar variables $x_1, \dots x_v$ which changes the form of the core polynomial to one that is linear in the oil variables because the terms associated with vinegar variables become constants. We are left with a linear system of o equations in o oil variables, $x_{v+1}, \dots, x_n$. It is then easy to solve the linear system using elementary row operations to reduce the associated matrix to echelon form and subsequently solve for the variables. The table below shows the terms and their types in core map F after the random assigning of vinegar variables.

Quadratic terms	none
Linear terms	o, $v \cdot o$
Constant terms	λ, $v \cdot v$, v

Since the values for the vinegar variables are chosen at random there is a high chance that a solution to the system will be obtained [3] however in the scenario that there is no solution to the resulting system after the random assignment of the variables, we simply start again and reassign different random values and this process continues until a solution is found. The resulting pre-image $l = (x_1, \ldots, x_n)$ is then used during the signature generation process where the signature is given by $t = T^{-1}(l)$. This process of inverting F to calculate the pre-image of the hash value under F is encapsulated in an algorithm provided below.

Inversion of core map.
Input: Core map F with random coefficients, hash value z.
Step 1: Randomly assign values to vinegar variables, $x_1, \ldots, x_v$ in all f_i
Step 2: Gaussian eliminate resulting linear system
Step 3: Solve linear system in oil variables $(x_{v+1}, \ldots, x_n)$
Step 4: If no solution go back to step 1 and start again **else** continue
Output: Pre-image: $(x_1, \ldots, x_n)$

4.2 The rainbow scheme

A variation of the UOV scheme known as the **Rainbow Scheme** was proposed by Ding and Schmidt in 2005 [16]. The rainbow scheme is based on the UOV scheme except it uses a stacked UOV configuration in which two (or more) UOV schemes are used to create a layered scheme, hence the name rainbow scheme (there are different layers of colour in a rainbow). This is best described with a small example. Consider the Rainbow Scheme given by the parameters (24, 12, 12) which stands for $v = 24$ vinegar variables, $o = 12$ oil variables and there are 12 oil-vinegar polynomials. We will consider an example where we stack two UOV schemes. For the first layer we will define a UOV system in which the core map is F_1 and $o_1 = 12, v_1 = 24$ and there will be 12 oil-vinegar polynomials. So F_1 will look like the following: $F_1 = (f_1(x_1, \ldots, x_{36}), \ldots, f_{12}(x_1, \ldots, x_{36}))$ and here we have $(x_1, \ldots, x_{24})$ vinegar variables and $(x_{25}, \ldots, x_{36})$ oil variables, so this layer is just a usual UOV scheme with the parameters given above. Now for the second layer we will define another usual UOV scheme this time the value of $n = v_1 + o_1$ from the first layer will be used as the number of vinegar variables for this layer, and we will have 12 additional oil variables in the layer also. So for the second layer, $o_2 = 12, v_2 = 36$ and there will again be 12 oil-vinegar polynomials. So the core map of the second layer will look like: $F_2 = (f_{37}((x_1, \ldots, x_{48}), \ldots, f_{48}(x_1, \ldots, x_{48}))$ and here we have $(x_1, \ldots, x_{36})$ are the vinegar variables and $(x_{37}, \ldots, x_{48})$ are the oil variables.

In the rainbow scheme the public key is given by $P = T_1 \circ F \circ T_2$ where $F = (F_1, F_2)$ so the core map of the rainbow scheme is composed of the core maps of two usual UOV schemes, which are then stacked. What we end up with, is 24 polynomials, 12 from each layer. So the scheme is highly efficient since signature generation only requires the solving of two simple linear systems

each with 12 variables and 12 equations in the same manor described above for usual UOV. It is also claimed that a nice feature of the rainbow scheme is that it is resistant to side attacks since there are two transformations either side of F which hide its structure. An observation that can be made is that one of the main goals of an attacker when attacking a Rainbow scheme could be to separate F into its constituent parts F_1 and F_2 as this would clearly make breaking the scheme far easier. It turns out that solving something called the MiniRank problem could be used to do this [16], in short the MiniRank problem is just trying to achieve the minimal rank of a set of matrices using linear combinations, this will be discussed in more detail in section 5 of this paper. However, the key point here is that this MiniRank problem is also a very hard problem and rank minimization is known to be NP-Complete [17] which is good news for the rainbow scheme.

5 Algebraic techniques used in cryptanalysis

There are a variety of algebraic techniques employed in cryptanalysis, in this section a detailed explanation of the main methods used will be given, as well as a mathematical understanding of how these methods work from the cryptographic point of view. We start by considering the direct attack, which tries to directly solve the polynomial system $P = H(q)$, where $H(q)$ is the hash of the document q as described earlier.

5.1 Grobner bases

Grobner bases calculations are a very powerful tool when working with polynomials in multiple variables. The theory of Grobner bases was created by Bruno Buchberger in his Ph.D thesis [18]. As stated above a direct attack involves directly solving the system $P = H(q)$ in order to break the scheme. Computing a Grobner basis is one such method. The general set-up is as follows.

More detail will be explained about the process of Buchberger's Algorithm [19] later on but the purpose of this diagram is simply to show that given a set of input polynomials $P = (p_1, \ldots, p_n)$ which will be the multivariate system of equations in the case of MPKC's, one can generate a set of output polynomials $G = (g_1, \ldots, g_n)$ (using Buchberger's algorithm) known as the Gronber basis. The output G provides information about the input set of polynomials P. In cryptanalysis this is useful as it allows an attacker to use the Grobner basis to answer questions about the polynomial system that forms the public key (the input). A very simple example follows in order to demonstrate the use of a Grobner basis. Given the input set of polynomials in variables x, y, z:

$(6x + 6y + 4z - 11, 7x + 9y + 11z - 24)$ we compute the Grobner basis of this system to be the set G given by $G = [x - \frac{5}{2}z + \frac{15}{4}, y + \frac{19}{6}z - \frac{67}{12}]$. Upon analysis of G we can deduce that the system has a solution space in one dimension, there are an infinite number of solutions for x, y, z, and the solutions are lines in 3 dimensions on the x,y,z-axes. In this example we have seen how informative the Grobner basis is, and how we can more easily establish a lot of information about the initial system by analysing the Grobner basis as opposed to the initial system itself. We will now go into detail about how the Grobner basis is computed and its use in cryptanalysis as well as providing a mathematical understanding of the algebra used during the process.

5.1.1 Computing a Grobner basis

Before explaining how a Grobner basis can be calculated and used from the point of view of cryptography, we first give a series of mathematical definitions that underpin the process of computing a Grobner basis and it's subsequent theory.

Definition. A **term ordering** is a total order $\prec$ on the set of monomials $A_1, A_2, \ldots, A_n$ if a) $\prec$ is multiplicative: $A_1 \leq A_2 \Rightarrow A_1 \cdot A_k \leq A_2 \cdot A_k$ and b) the monomial 1, is the smallest [20].

Definition. An ideal generated by a set of polynomials P in $K[x_1, \ldots, x_n]$ (the polynomial ring over K) is the set $\langle P \rangle = \{d_1 p_1 + \cdots + d_n p_n | p_i \in P, d_i \in K[x_1, \ldots, x_n]\}$ i.e the spanning set of P, which is the set of all linear combinations of the elements in P, with polynomial multipliers d_i forming polynomial coefficients.

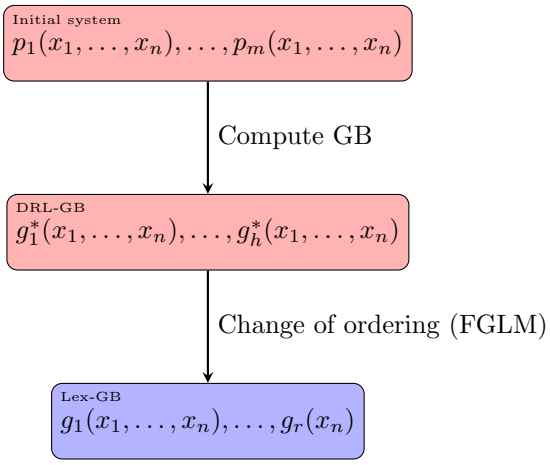
Definition. The variety of a set of polynomials P, also known as the zero set, is the set of all complex solutions to the given set of polynomials. i.e $V(P) = \{(s_1, \ldots, s_n) | (s_1, \ldots, s_n) \in \mathbb{C}^n, P(s_1, \ldots, s_n) = 0 \text{ for all } p \in P\}$

An attacker wishes to solve the non-linear system of equations that form the cryptosystem, and computing the Grobner basis of the system allows the attacker to recover the variety of the ideal generated by the system, hence obtaining a solution set and breaking the scheme. An observation we make here is that any cryptosystem can essentially be described as a system of nonlinear multivariate polynomial equations (not just MPKC's). We call the initial system P where $P = p_1(x_1, \ldots, x_n), \ldots, p_m(x_1, \ldots, x_n)$ and each p_i is a multivariate polynomial.

It is important to highlight how the term ordering affects the computation of a Grobner basis, the variety can be recovered from a lexicographic Grobner basis, however in practice the Lex-GB is never actually calculated directly since this is considered to be computationally bad [21]. Instead a degree reverse lexicographic ordering is used in order to calculate a DRL-GB first, upon which a

change of ordering algorithm is applied [21] which converts the DRL-GB to a Lex-GB. This algorithm known as the FGLM algorithm [22] is a polynomial time algorithm with complexity $O(nD^3)$ where the polynomials are in n variables, and D is the degree of the ideal generated by by the system of polynomials. It is easier to compute a Lex-GB this way as opposed to computing a Lex-GB directly. This method means the Lex-GB Grobner basis is not actually being recalculated from the DRL-GB, but instead the ordering is just being changed which is computationally good.

As stated earlier the variety can be recovered from the Lex-GB, the variety of the Grobner basis is equal to the variety of the ideal generated by the initial polynomial system. So the generated Grobner basis is simply another basis of the ideal itself, and therefore has the same zero set (solution set). The general setup for the computation of a Lex-GB is now shown below.



The Lex-GB has a triangular shape of the following form:

$$\begin{aligned} & g_1(x_1, \ldots\ldots, x_n) \\ & \quad \ldots \\ & \quad \ldots \\ & \qquad g_r(x_n) \end{aligned}$$

Where $g_r(x_n)$ is a univariate polynomial, that is, a polynomial in one variable. The initial problem of solving a system of non linear multivariate polynomial equations has been reduced to solving a univariate polynomial in a single variable. By computing x_n all roots can be found by plugging the value for x_n into the polynomial above it which has two variables (x_n is one them and is now known) and continuing up in this fashion to yield the full solution set.

5.1.2 Buchberger's algorithm

The first algorithm that allowed for the computation of a Grobner basis was Buchberger's Algorithm [19]. Before stating the algorithm Buchberger's Criterion is given.

Buchberger's Criterion [23]. The set G is a Grobner basis if and only if for every $g_1, g_2 \in G$ the normal form of the corresponding S-polynomial they generate equals zero upon reduction using the multivariate division algorithm with respect to G.

Definition. The S-Polynomial (subtraction polynomial) generated by $g_1, g_2 \in G$ is $t_1 g_1 - t_2 g_2$ where t_1, t_2 are monomials such that product of t_1 and the leading term of g_1 is equal to the product of t_2 and the leading term of g_2.

Buchberger's Algorithm is shown below.

Buchberger's Algorithm.
Input: Set of polynomials P that generate the ideal I
Step 1: Apply Buchberger's Criterion to the set P, if true go to step 3 else continue
Step 2: We have found $f = \text{normalform}(t_1 g_1 - t_2 g_2) \neq 0$ so set $P = P \cup \{f\}$, go back to step 1
Step 3: Return Grobner basis = G.
Output: $G = P$, a Grobner Basis for the ideal I.

We point out that this algorithm always terminates as a consequence of Hilbert's Basis Theorem [24] which is that that every ideal in the polynomial ring K$[x_1, \ldots x_n]$ is finitely generated. This means there is a finite number of polynomials in P that generate the ideal $I = \langle P \rangle$ hence the increasing collection of ideals cannot be infinite and the algorithm will terminate. The complexity of computing these Grobner bases will now be detailed.

5.1.3 Macaulay Matrices

We now introduce the **Macaulay Matrix** which is at the centre of the complexity analysis of computing a Grobner basis. We consider the product of all the initial polynomials in the system of m equations with monomials t_i of degree d with respect to a monomial ordering $\prec$. We can then represent these polynomials as a vector and the coefficients of the vector are the coefficients of the monomials, which then form the Macaulay matrix. So the rows are the products of the polynomials p_i in the initial system and the monomials t_i i.e $p_i t_i$, and the columns form the monomial bases. Informally the process of computing a Grobner basis can be understood as performing Gaussian elimination on Macaulay matrices of increasing degree [25], so we are considering multiples of the initial polynomials of increasing degree. This process will terminate at some point at a Macaulay matrix of some degree, $D^{regularity}$, which is known

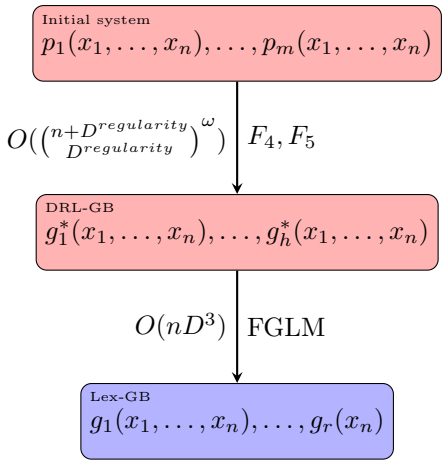
as the degree of regularity [26]. Gaussian elimination on this Macaulay matrix will produce a Grobner basis. It is clear that the degree of regularity is fundamental in understanding the complexity of computing a Grobner basis, since you can bound the complexity by the cost of performing Gaussian elimination on the largest Macaulay Matrix achieved (with degree = $D^{regularity}$) [25].

All of the above is calculating the Grobner basis with respect to the DRL ordering, the first step. The next step is then to then apply the FGLM change of ordering algorithm in order to obtain the Lex-GB in polynomial time and therefore the associated variety. Buchberger's Algorithm was the first algorithm that could compute a Grobner basis but it was was inefficient and hard to implement in polynomials with many variables, subsequently more efficient Grobner basis algorithms have been created based on the same method as Buchberger such as F_4 [13] and F_5 [27]. These algorithms do not improve the worst case complexity of Buchberger's algorithm however they are much faster when implemented. The complexity of F_5 is $O(\binom{n+D^{regularity}}{D^{regularity}}^{\omega})$ where $2 < \omega < 3$ is the linear algebra constant, but in practice the algorithm is much more efficient than Buchberger's as it removes useless reductions to 0 [27], consequently it is able to calculate a Grobner basis more efficiently. These algorithms have led to the breaking of the HFE scheme by computing the Grobner basis of the public system of polynomials [12], before them it was extremely difficult to compute a Grobner basis. An example illustrating this is given below using the Buchberger algorithm and improved Buchberger algorithm in sage to compute the Grobner basis of the ideal $(x+y+z, xy+xz+yz, xyz-1)$ over the ring $\mathbb{F}_{121}[x,y,z]$.

$(x+y+z, xy+xz+yz) \Rightarrow y^2+yz+z^2$
$P : (x+y+z, xy+xz+yz, xyz-1, y^2+yz+z^2)$
$(xy+xz+yz, x+y+z) \Rightarrow 0$
$P : (x+y+z, xy+xz+yz, xyz-1, y^2+yz+z^2)$
$(xyz-1, xy+xz+yz) \Rightarrow z^3-1$
$P : (x+y+z, xy+xz+yz, xyz-1, y^2+yz+z^2, z^3-1)$
$(x+y+z, y^2+yz+z^2) \Rightarrow 0$
$P : (x+y+z, xy+xz+yz, xyz-1, y^2+yz+z^2, z^3-1)$
$(xy+xz+yz, xyz-1) \Rightarrow 0$
$P : (x+y+z, xy+xz+yz, xyz-1, y^2+yz+z^2, z^3-1)$
$(x+y+z, xyz-1) \Rightarrow 0$
$P : (x+y+z, xy+xz+yz, xyz-1, y^2+yz+z^2, z^3-1)$
$(xy+xz+yz, y^2+y*z+z^2) \Rightarrow 0$
$P : (x+y+z, xy+xz+yz, xyz-1, y^2+yz+z^2, z^3-1)$
$(xyz-1, x+y+z) \Rightarrow 0$
$P : (x+y+z, xy+xz+yz, xyz-1, y^2+yz+z^2, z^3-1)$
$(xyz-1, y^2+yz+z^2) \Rightarrow 0$
$P : (x+y+z, xy+xz+yz, xyz-1, y^2+yz+z^2, z^3-1)$
$(x+y+z, z^3-1) \Rightarrow 0$
$P : (x+y+z, xy+xz+yz, xyz-1, y^2+yz+z^2, z^3-1)$

$(xyz-1, z^3-1) \Rightarrow 0$
$P : (x+y+z, xy+xz+yz, xyz-1, y^2+yz+z^2, z^3-1)$
$(xy+xz+yz, z^3-1) \Rightarrow 0$
$P : (x+y+z, xy+xz+yz, xyz-1, y^2+yz+z^2, z^3-1)$
$(y^2+yz+z^2, z^3-1) \Rightarrow 0$
$G : (x+y+z, xy+xz+yz, xyz-1, y^2+yz+z^2, z^3-1)$

In the above calculation of the Grobner basis G using Buchbergers algorithm it can be seen that there are 11 useless reductions to zero. Now using the improved Buchberger algorithm the calculation performs no useless reductions to 0 in order to compute the Grobner basis $G = (x+y+z, y^2+yz+z^2, z^3-1)$. So it is clear that in practice the improved algorithms perform much more efficiently. Finally the complete process of computing a Grobner basis is illustrated below.



5.2 XL Algorithm

The XL algorithm [28] works generally as follows. Given a non-linear system P, all equations in P are multiplied by all monomials of a particular degree D preserving the solutions of P, this is known as extending the system. Eventually, the degree D will be large enough to yield around the same number of linearly independent equations and monomials. System P can then be solved by linearization of the extended system (performing Gaussian elimination) in order to produce a univariate polynomial. This univariate polynomial is then solved e.g by using the Berlekamp Algorithm [11] in order to recover the solution of one of the variables. The value of this variable is then substituted into the system and the process is repeated until the full solution set is obtained. This algorithm can used to directly attack cryptosystems or instead be used as part of other cryptanalysis techniques as a means of solving systems of equations. It is important to note that the XL algorithm can used to solve overdetermined systems of

equations. There are also many variants of the XL algorithm that improve efficiency such as FXL [28] and XL2 [29]. A system is said to be determined if the number of variables is equal to the number of equations, underdetermined if the number of variables is more than the number of equations and overdetermined if the number of variables is less than the number of equations. The algorithm [28] is given below.

XL Algorithm.
Input: System of polynomials $p_1, \ldots, p_n$.
Step 1: Multiply all equations $p_1, \ldots, p_n$ by all monomials of particular degree $\leq D$ (**Extending the system**).
Step 2: Gaussian eliminate resulting extended system to generate a univariate polynomial (**Linearization**).
Step 3: Solve the univariate equation (using Berlekamp's Algorithm) to obtain the value of the associated variable.
Step 4: Substitute this value into the system and simplify, and **repeat** with this resulting system until you have solutions for all the variables in the original system.
Output: Solution set for all variables in the original system $p_1, \ldots, p_n$.

5.3 Structural attacks

Structural attacks are also useful in cryptanalysis and they take a different approach to the direct attack described above. Such attacks are generally focused on computing an equivalent private key which may be used by an attacker to decrypt and forge signatures. One such attack was one carried out by Kipnis and Shamir [9] on the original oil and vinegar scheme created by Patarin [2]. The balanced oil and vinegar scheme [2] in which the number of oil and vinegar variables are the same, was defeated by Kipnis and Shamir in an attack that breaks the scheme in polynomial time. An overview of this attack will now be given.

5.3.1 Kipnis-Shamir attack on balanced Oil and Vinegar (OV)

We first define two subspaces of the vector space $\mathbb{F}^n$ and an oil-vinegar matrix.

Definition. The Oil subspace $\mathcal{O}$ is the set of all vectors in $\mathbb{F}^n$ such that all the vinegar variables are equal to 0.

Definition. The Vinegar subspace $\mathcal{V}$ is the set of all vectors in $\mathbb{F}^n$ such that all the oil variables are equal to 0.

Definition. If $\hat{p}(x_1, \ldots, x_n)$ represents the homogeneous quadratic part of the public key $p(x_1, \ldots, x_n)$ then $\hat{p}(x_1, \ldots, x_n)$ can be written as $\hat{p}(x_1, \ldots, x_n) = (x_1, \ldots, x_n) \cdot B \cdot (x_1, \ldots, x_n)^T$ where B is known as the **oil-vinegar matrix** and $B = \left(\begin{smallmatrix} * & * \\ * & 0 \end{smallmatrix}\right)$ where $*$ represents $v \times v$, $v \times o$, $o \times v$ terms and the 0 represents

the absence of $o \times o$ terms, and here the T represents the transpose.

The attack of Kipnis and Shamir [9] is shown using a homogeneous version of the OV scheme for the sake of simplicity, but in their paper they described how the same attack can be used for the non homogeneous case [9]. Consider an oil-vinegar matrix $\left(\begin{smallmatrix} * & * \\ * & 0 \end{smallmatrix}\right)$ and denote this by H. Then taking a random value $o \in \mathcal{O}$ and forming the matrix $o\left(\begin{smallmatrix} * & * \\ * & 0 \end{smallmatrix}\right) = H \cdot o$, we see that $H \cdot o \in \mathcal{V}$ as all the oil variables are equal to 0. Next upon inversion of the matrix H it follows that $H^{-1} \cdot v \in \mathcal{O}$ as all the vinegar variables are equal to 0. Now we consider two oil-vinegar matrices H and F and combining the two above results we obtain $(F^{-1} \cdot H) \cdot o \in \mathcal{O}$ which means the image of the value of the oilspace is also in the oilspace $\Rightarrow \mathcal{O}$ is an invariant subspace of $F^{-1} \cdot H$. When we translate this to the public key, we denote the homogeneous quadratic part of the public key polynomials $p_1, \ldots, p_n$ by the matrix L_i which gives us $L_i = S^T \cdot H_i \cdot S$, where H_i are the oil-vinegar matrices and S is the matrix illustrating the affine transformation $\mathcal{S}$. Then we take $o \in \mathcal{O}$ and $v = S^{-1}(o)$ to obtain the following:

$$\begin{aligned}(L_k^{-1} \cdot L_i) \cdot v &= (S^{-1} \cdot H_k^{-1} \cdot (S^T)^{-1} \cdot S^T \cdot H_i \cdot S) \cdot S^{-1}(o) \\ &= S^{-1} \cdot H_k^{-1} \cdot H_i \cdot S \cdot S^{-1}(o) \\ &= S^{-1}(o) \cdot (H_k^{-1} \cdot H_i) \in S^{-1}(o)\end{aligned} \tag{4}$$

$\Rightarrow$S$^{-1}(o)$ is an invariant subspace of $(L_k^{-1} \cdot L_i)$
Then we randomly choose a value $k \in \{1, 2, \ldots, o\}$ such that the matrix L_k is invertible and calculate the product $(L_k^{-1} \cdot L_i)$ then calculate the invariant subspaces which allows us to find an equivalent affine transformation $\mathcal{S}^*$ which is then used to find an equivalent core map $\mathcal{F}^* = \mathcal{P} \circ (\mathcal{S}^*)^{-1}$ hence breaking the scheme. It should be noted that this attack defeats the oil-vinegar scheme in the case where $v = o$ in polynomial time [9] and also when $v \leq o$ however, when $v > o$ the complexity of this attack is approximately $q^{v-o} \cdot o^4$ [30] so choosing $v \approx 2o$ renders this attack not very useful in the unbalanced oil-vinegar case as a consequence of the resulting high complexity.

5.3.2 MiniRank attack

Another structural attack that tries to generate an equivalent private key is the so called MiniRank attack which is trying to solve the MiniRank rank problem [31]. This attack utilises the fact that the associated matrices of the core maps of many multivariate schemes have a low rank. We introduce the MiniRank problem, which is to find a linear combination of a collection of matrices, which is of low rank.

MiniRank Problem. Given k nxn matrices A_i for $i = 1, \dots k$, compute a linear combination $G = \sum_{i=1}^{k} \alpha_i A_i$ such that the Rank$(G) \leq r$ [31].
We look at linear combinations of the of the matrices associated with the public key polynomials which have low rank. Such a linear combination therefore corresponds to a core equation, and this then leads to an equivalent affine map S and by further analysis we can recover an equivalent map T and F which ultimately forms an equivalent private key which can be used to decrypt messages or forge signatures. In the case of HFE there is also a Kipnis-Shamir MiniRank attack [32], this is a structural attack which involves lifting the map back up to the big field. Let $\hat{F}(x) = \sum_{q^i+q^j \leq D} \alpha_{ij} X^{q^i+q^j}$ be the quadratic form of the associated core map F and let $\hat{F} = [\alpha_{ij}]$ be the associated matrix of $\hat{F}$. We can make a basic observation that a fundamental characteristic of this polynomial is that the rank of the associated matrix is low since the degree of F is controlled. Kipnis and Shamir demonstrated that if we can solve the associated MiniRank problem then we can break HFE. In short, if degree of F is low (as is required by HFE) this implies that the rank of the associated matrix is low, and this means that the MiniRank attack could work and this can be used to generate equivalent keys.

When new schemes are being designed avoiding the MiniRank attack is something that needs to be considered. The ZHFE scheme described earlier is a good example of how avoiding the MiniRank attack can be done in terms of HFE. As described, the degree of the two core maps in ZHFE $F(x)$ and $\tilde{F}(x)$ are kept very high [14], this implies the rank of the associated matrices are also very high and so avoids the MiniRank problem. Other attacks include the high rank attack and differential attacks which both yield information about the private key.

6 My Contribution.

As part of this research project we have implemented the three main algorithms for the standard UOV scheme and the lifted UOV scheme [4] in sage. The three algorithms are key generation, signature generation and signature verification. As detailed in their paper [4], we consider UOV schemes over a finite field of characteristic 2. The main idea of the new scheme is to form the public key over $\mathbb{F}_2$ and then lift it to a bigger field $\mathbb{F}_{2^r}$, the purpose of this is that forming the keys over $\mathbb{F}_2$ results in smaller public keys and using these smaller keys in a UOV scheme over $\mathbb{F}_{2^r}$ preserves the difficulty of solving the MQ-Problem over a bigger field [4] where variables can take more values. The three algorithms to create a usual UOV scheme over $\mathbb{F}_{2^r}$ are:

Key generation.
Input: Finite field $\mathbb{F}_{2^r}$, integers v and o, fix $n = v + o$.
Step 1: Randomly generate core map $F : \mathbb{F}_{2^r}^n \to \mathbb{F}_{2^r}^n$
Step 2: Randomly generate transformation $T : \mathbb{F}_{2^r}^n \to \mathbb{F}_{2^r}^n$
Step 3: Form the public key $P = F \circ T$.
Output: Public key $P = F \circ T$ and private key (F, T)

Signature generation.
Input: Private key (F, T), hash value, z, of document, q, to be signed: $z = H(q)$.
Step 1: Apply **Inversion of the core map** algorithm to F to calculate pre image $\mathbf{x} = (x_1, \ldots, x_n)$ of z under F.
Step 2: Compute signature: $t = T^{-1}(\mathbf{x})$.
Output: Signature t.

Signature verification.
Input: Public key $P = F \circ T$, signature t, hash value z.
Step 1: Evaluate t under P: $z' = P(t)$
Step 2: Check equality holds: $z = z'$?
Step 3: **If** true accept t **else** reject.
Output: Accepted signature t or signature rejection.

The only difference between the standard UOV set up and the new scheme, is that in the above key generation algorithm the finite field used is $\mathbb{F}_2$ instead of $\mathbb{F}_{2^r}$. The remaining algorithms, signature generation and signature verification, take place over the large field as normal [4].

6.1 Generating the public key

To illustrate this, the block of sage code used to implement the public key generation algorithm in the normal UOV scheme is shown below.

```
for i in range(o):
    D[i] = random_matrix(K,v,v) #the vv part of the ov matrix.

for i in range(o):
    L[i] = random_matrix(K,v,o) #the vo part of the ov matrix.

for i in range(o):
    I[i] = random_matrix(K,o,v) #the ov part of the ov matrix.

for i in range(o):
    M[i] = matrix(K,o,o)        #the oo part of the ov matrix.

for i in range(o):
    AO0[i]=D[i]
    AO1[i]=L[i]
    AO2[i]=I[i]
```

```
    A03[i]=M[i]

for i in range(o):
    N[i] = matrix.block([[A00[i],A01[i]],[A02[i],A03[i]]])
    #the oil vinegar matrix.

F=[0 for i in range(o)]
print "Generating Oil-Vinegar matrices..."
for i in range(o):
    F[i]=N[i] #the core oil vinegar matrix.
    print F[i]
    print""
    tp1*F[i]*tp1.transpose() #the oil vinegar polynomials.
    print ""

while true:
        T=random_matrix(K,n) #transformation T
        if T.is_invertible():
            break
Tt=T.transpose()

P=[0 for i in range(o)]
print "Generating public key polynomials... "
for i in range(o):
    P[i]=Tt*F[i]*T
    print tp1*P[i]*tp1.transpose() #public key polynomials.

print ""
```

Where K is the big field and is defined by $K\langle a\rangle = GF(q)$ where $q = 2^r$ and tp1 the matrix of the generators of the polynomial ring Pp=PolynomialRing(K, x, n) for example if $n = 6$ then tp1 $= (\ x0\ x1\ x2\ x3\ x4\ x5\ x6\ )$, and tp1.transpose() is the corresponding transpose of this matrix. We can see that in this construction the coefficients in each of the 4 parts D[i], L[i], I[i] and M[i] making up the oil vinegar matrix are from the big field K and the linear transformation also has coefficients in K. From this it follows that the generated public key has coefficients in K, as is expected in the normal UOV scheme. The output of this code is shown for $K\langle a\rangle = \mathrm{GF}(2^2)$, $o = 2$, $v = 4$ and $n = 6$.

$$\left(\begin{array}{cccc|cc} 0 & 0 & a+1 & 0 & a+1 & a \\ a & 1 & 0 & a & 0 & 0 \\ a & 0 & a+1 & 1 & 1 & a \\ 1 & 0 & 0 & 1 & a+1 & a+1 \\ \hline a+1 & 0 & a+1 & 1 & 0 & 0 \\ a & a & 1 & a & 0 & 0 \end{array}\right)$$

$(ax_0x_1 + x_1^2 + x_0x_2 + (a+1)\,x_2^2 + x_0x_3 + ax_1x_3 + x_2x_3 + x_3^2 + ax_2x_4 + ax_3x_4 + ax_1x_5 + (a+1)\,x_2x_5 + x_3x_5)$

$$\left(\begin{array}{cccc|cc} a+1 & 0 & 0 & 1 & a & a \\ 0 & a+1 & a+1 & 1 & 1 & a \\ 1 & a & 1 & a & a & a \\ a & 1 & a & 0 & a+1 & 1 \\ \hline a+1 & a & 1 & a+1 & 0 & 0 \\ a & a & 0 & 1 & 0 & 0 \end{array}\right)$$
$(a+1)\,x_0^2+(a+1)\,x_1^2+x_0x_2+x_1x_2+x_2^2+(a+1)\,x_0x_3+x_0x_4+(a+1)\,x_1x_4+(a+1)\,x_2x_4+ax_2x_5)$

Public key:
$(ax_0^2+x_0x_1+ax_1^2+x_0x_2+(a+1)\,x_1x_2+x_2^2+(a+1)\,x_0x_3+(a+1)\,x_1x_3+(a+1)\,x_2x_3+(a+1)\,x_3^2+x_0x_4+ax_2x_4+(a+1)\,x_4^2+x_0x_5+ax_1x_5+x_2x_5+ax_3x_5+(a+1)\,x_4x_5+ax_5^2)$
$(ax_0^2+(a+1)\,x_1^2+x_0x_2+(a+1)\,x_1x_2+(a+1)\,x_2^2+ax_0x_3+x_1x_3+x_2x_3+ax_3^2+(a+1)\,x_1x_4+(a+1)\,x_2x_4+ax_3x_4+x_4^2+ax_1x_5+ax_2x_5+(a+1)\,x_3x_5+x_4x_5+(a+1)\,x_5^2)$

This shows the oil-vinegar matrices and the associated oil vinegar polynomials in the new scheme when $o = 2$. Where a is an element in the finite field K. The output shows 2 oil vinegar matrices (since the number of oil variables was chosen to be equal to 2) each with a corresponding oil vinegar polynomial directly beneath it. An important observation here is to notice the block of 0's in the bottom right block of each oil-vinegar matrix, this corresponds to the missing oil-oil terms in the core map. Furthermore, the coefficients of the all the maps are in the field K. The public key is then given by the composition of of the core map F (the 2 oil vinegar polynomials) with the transformation T which is an nxn invertible matrix, to yield the two multivariate equations forming the public key. We again point out that the public key is formed by the composition of the core map and the linear transformation. This composition is done by first realising that in matrix notation we have $T(\tilde{x}) = T\tilde{x}$ where the vector $\tilde{x}^T = (x_1, \ldots, x_n)$ (where the power of T represents the transpose). Considering each oil-vinegar matrix given by $F[i]$ that form the core map, and the transformation T we arrive at the following: $F \circ T = (T\tilde{x})^T F\big[i\big](T\tilde{x}) = \tilde{x}^T(T^T F\big[i\big]T)\tilde{x}$. This corresponds to the lines

```
P[i]=Tt*F[i]*T
print tp1*P[i]*tp1.transpose() #public key polynomials.
```

in the the above code, and in this case $\tilde{x}^T$ = tp1 = $(x_0, \ldots, x_5)$ and P[i] corresponds to $T^T F\big[i\big]T$. This explains how the key generation has been implemented for normal UOV, but to implement the same algorithm for the new scheme we had to make some changes which will be detailed now. Recall that the new scheme requires the public key to be made over the field $\mathbb{F}_2$, we first defined such a field $K2$ as follows:

```
K2.<a>=GF(2)
```

The process of generating the keys for the new scheme is identical to that of the old scheme shown above except that each part that forms the oil vinegar matrix is instead created over $K2$, and not K as was the case before. This is illustrated in the following block of sage code:

```
for i in range(o):
    D[i] = random_matrix(K2,v,v) #the vv part of the ov matrix.

for i in range(o):
    L[i] = random_matrix(K2,v,o) #the vo part of the ov matrix.

for i in range(o):
    I[i] = random_matrix(K2,o,v) #the ov part of the ov matrix.

for i in range(o):
    M[i] = matrix(K2,o,o)        #the oo part of the ov matrix.
```

This means the corresponding core map F has coefficients in $K2$. The transformation T is also required to have coefficients in $K2$ to allow it to be composed with F and so for the new scheme the matrix T was defined over K2 instead of K:

```
    while true:
        T=random_matrix(K2,n) #transformation T
        if T.is_invertible():
            break
Tt=T.transpose()
```

The generation of the public key is then the same as that of a normal UOV scheme described above except with a subtle difference. We recall that key generation in the new scheme occurs over $\mathbb{F}$ but the subsequent public key is used as the public key of a UOV system over $\mathbb{F}_{2^r}$. So the public that we will have made in the new scheme has been made over the small field $K2$, but we need to view it over the large field K so that the subsequent signature generation and signature validation algorithms can still take place over the large field K (as is required by the new scheme). We do this by utilising a feature in sage that allows us to change the ring of a matrix, and this is done as we take the composition of the core map with T, as is shown in the following block of code:

```
P=[0 for i in range(o)]
print "Generating public key polynomials... "
for i in range(o):
    P[i]=Tt*(F[i].change_ring(K))*T
    print tp1*P[i]*tp1.transpose() #public key polynomials.
```

We end up with a public key with coefficients in the field $K2$, that is viewed over the field K. It is necessary to make this change as the subsequent algebra used in the signature generation and validation algorithms of the new scheme

would have caused problems in sage if the public key was left over $K2$, since these algorithms are executed over K. To make sure that the public key has indeed been viewed over the big field K, sage has a feature that allows us to check the parent ring of of the public key, and was used to do so as follows:

```
parent(P[i])
```

which outputs the following (for the case of K =GF(2^2) and $n = 6$): Full MatrixSpace of 6 by 6 dense matrices over Finite Field in a of size 2^2. This confirms that the ring has been indeed been changed to the big field. A small example of the output of the code for generating the public key in the new scheme is now shown with parameters $K = \mathbb{F}_{2^2}$, $o = 2, v = 4$ and $n = 6$ as before.

$$\left(\begin{array}{cccc|cc} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{array}\right)$$

$$\left(x_0^2 + x_0x_1 + x_1^2 + x_1x_3 + x_1x_4 + x_3x_5\right)$$

$$\left(\begin{array}{cccc|cc} 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{array}\right)$$

$$(x_2x_3 + x_0x_4 + x_1x_4 + x_1x_5 + x_3x_5)$$

Public key:

$$\left(x_0^2 + x_0x_1 + x_3^2 + x_0x_4 + x_1x_4 + x_4^2 + x_1x_5 + x_2x_5 + x_4x_5 + x_5^2\right)$$

$$\left(x_1x_3 + x_3^2 + x_1x_4 + x_2x_4 + x_3x_4 + x_0x_5\right)$$

Where the matrices are the oil-vinegar matrices and each oil vinegar matrix is proceeded by its corresponding oil vinegar polynomial, and lastly the resulting public key is shown. We can again make the observation that there is a block of 0's in the bottom right block of each of the oil-vinegar matrices and this corresponds to the missing oil-oil terms in the construction of the core map. Notice also that all the coefficients of the oil-vinegar matrices, the oil-vinegar polynomials and the public key polynomials are in the small field $K2$ as required. A comparison can be made about the difference in size of the two public keys from both schemes, clearly the public keys of the new scheme are smaller since only one bit is needed for each coefficient whereas in the UOV scheme over $\mathbb{F}_{2^r}$ there would be r bits needed for each coefficient [4]

6.2 Signature generation

The next algorithm we implemented was the signature generation algorithm, as before we will first show the implementation of the algorithm for the regular UOV scheme over $\mathbb{F}_{2^r}$ and then highlight the changes that were made to make it work for the new field lifted scheme. To implement the signature generation algorithm for a UOV scheme over $\mathbb{F}_{2^r}$ we recall that the first stage of generating a signature is to calculate the pre-image of the hash value under the core map F. We refer back to the algorithm introduced earlier in the paper that is used to calculate such a pre-image. The first step of this algorithm is to randomly assign values to the vinegar variables yielding a linear system of equations in the oil variables, then Gaussian elimination is used to recover the solutions of the remaining oil variables hence yielding the pre-image $(x_1, \ldots, x_n)$. First we need to assign the random values to the vinegar variables and this is shown in the following block of sage code:

```
coremap = [tp1*F[i]*tp1.transpose() for i in range(o)]
print "Core map F..."
for i in range(o):
    F[i]=N[i]
    print tp1*F[i]*tp1.transpose()
Ry=PolynomialRing(K,n-v,['x%s'%p for p in[v..n-1]])
images = [K.random_element() for i in range(v)]+list(Ry.gens())
phi=Pp.hom(images,Ry)
List=[coremap[j] for j in  range(o)]
coremap_subs = [phi(f[0]) for f in List]
print ""
```

In the above code we can see that evaluation of an expression exhibits a ring homomorphism, and we construct a homomorphism that takes the first v variables $(x_1, \ldots, x_v)$ from the polynomials that make up the core map and assigns random elements from the field K to them but retains the unknown oil variables $(x_{v+1}, \ldots, x_n)$, which in this case come from the generators of the ring Ry. Once the homomorphism phi has been applied to all the equations in the core map, the vinegar variables have random values from K and the oil variables remain unknown which leaves a linear system in the oil variables. So at this stage we currently know the values for the vinegar variables $(x_1, \ldots, x_v)$. The resulting system of o equations is made equal to the hash value which also has o elements by construction (random hash function used for simplicity). The next stage in signature generation is to solve the resulting linear system to get solutions for the oil variables. We implement this using the following segment of code:

```
s = [coremap_subs[i]-hashresult[i] for i in range(o)]
h = Ry.ideal(s)
def give_result(ring,coremap):
    h = ring.ideal(coremap)
```

```
    print "Solving for oil variables..."
    result = h.variety()[0]
    print(result)
    print("")
    for i in [v..n-1]:
        images[i] = result['x'+str(i)]
    print "Pre-image of the hash value under the core map F..."
    print images
    return images

pre_image = give_result(Ry,s)
preimage = vector(pre_image)
```

We solve the linear system in the oil variables using the variety operation in sage that recovers the variety of the ideal generated by the linear system, which yields the zero set for the original system hence giving us solutions for the oil variables $(x_{v+1}, \ldots, x_n)$. To do this we defined a method, give result, that takes in a as parameters, a polynomial ring, and a core map which in this case is a list of the oil-vinegar polynomials. This method simply takes the set of equations defined by the list s which is just the same linear system of oil variables but with the hash values taken over to the left hand side of the equations so each equation is equal to zero, and solves for the oil variables $(x_{v+1}, \ldots, x_n)$ using the variety operation described above. At this stage, we now have a full pre-image of the hash value under the core map F, which is given by $(x_1, \ldots, x_n)$. Since signature generation in the new scheme is also done over the big field, this algorithm is implemented in the new scheme in exactly the same way as it is shown above, implemented in the usual UOV scheme. The obvious difference however, is that in the new scheme all of the coefficients of the oil-vinegar polynomials are in the small field $K2$, and this means that once we substitute in random values for the vinegar variables from the big field K, the vinegar variables will not only have values in the small field $K2$, but will take values from the big field K, and therefore the solutions for the values of the remaining oil variables will also be in the big field K which is exactly what is is intended by the new scheme. A small example of this is given by the output of the above code for the new scheme shown below, note that the parameters chosen in these examples are kept small for the sake of simplicity ($K = \mathbb{F}_{2^2}, o = 2, v = 2, n = 6$) and a is an element of the finite field K:

Core map F...
$[x_0^2 + x_0x_1 + x_2^2 + x_3^2 + x_2x4 + x_0x_5]$
$[x_1^2 + x_0x2 + x_1x_2 + x_2^2 + x_2x_3 + x_3^2 + x_1x_4 + x_2x_4 + x_3x_4 + x_0x_5 + x_1x_5]$

Hash result:
$[1, 7]$

Solving for oil variables...

$x_5 : a + 1, x_4 : a$

Pre-image of the hash value under the core map F...
$[a, 0, 0, a, a, a + 1]$

We now are left with the final stage of signature generation which will yield the actual signature itself. All that remains is to take the inverse of the transformation T with respect to the pre-image calculated from the from the last stage. This is simply done by multiplying the inverse of the matrix representing T by the pre-image, since we have that $T^{-1}(\mathbf{x}) = T^{-1}\mathbf{x}$ for a given pre-image $\mathbf{x} = (x_1, \ldots, x_n)$ as follows:

```
signature = T.inverse()*preimage
```

The output of this using the same pre-image and transformation T from the last example yields the signature: $(0, 1, a + 1, a + 1, 0, 0)$ where a is again an element of the big field.

6.3 Signature verification

The final algorithm to be implemented is the signature verification algorithm. Recall that in order to verify a signature candidate, we need to evaluate the signature under the public key P and then check that this is equal to the corresponding hash value of the document to be signed. In order to do this we need to take our signature candidate which is of the form $(x_1, \ldots, x_n)$ where each $x_i \in K$, and substitute these values into the corresponding variables of the public key. We do this by again forming a ring homomorphism, this time from the multivariate polynomial ring in the n variables over the finite field in a of size 2^r, to the finite field in a of size 2^r. This allows us to take the values of the signature and uses these values as the values of the corresponding variables in the public key. Finally a simple equality check is made between the hash value and the image of the signature under the public key as follows:

```
publickey = [tp1*P[i]*tp1.transpose() for i in range(o)]
images = [signature[i] for i in range(n)]
phi=Pp.hom(images,K)
myList=[publickey[j] for j in  range(o)]
publickey_subs = [phi(f[0]) for f in myList]
print "Signature validation..."
publickey_subs == hashresult
all(itertools.imap(lambda x: x in publickey_subs, hashresult))
```

Note that the final line is just a secondary equality check that checks that all the elements in the list publickey_subs are indeed elements of the list hashresult and since the lists are identical in size this is another equality check. The outputs of both lines are the booleans true or false (this implementation has been made to produce accepted signatures only). Finally it is clear that in the new scheme

since signature verification is done over the big field K as normal, this algorithm is implemented in exactly the same way as described above the only difference being that the public key in the new scheme has been made over the small field $K2$ ($\mathbb{F}_2$). One observation we make here is that in the new scheme although the public key was built over the small field $K2$, we are now viewing it in the large field K ($\mathbb{F}_{2^r}$) (as the algorithm takes place over the large field) so the ring morphism is still from the multivariate polynomial ring in the n variables over the finite field in a of size 2^r, to the finite field in a of size 2^r as required. Examples of the full outputs of each of these schemes are provided at the end of this report.

7 Discussion

We feel we have contributed to the understanding of multivariate cryptography and to the understanding of the new scheme with our report and implementations. The implementations show the stages involved in constructing each of the schemes and serve their purpose to give the reader a closer look at how multivariate schemes are constructed and the calculations involved. There were a number of challenges during the completion of this project, learning how to code in sage whilst simultaneously spending a lot of time researching and learning advanced mathematical concepts outside of our undergraduate level, proved challenging but just as rewarding.

One of the difficulties we faced was the implementation of field lifting in sage. We required the public key of the new scheme to be made in the small field but viewed in the large field, initially just making the key over the small field as usual and then trying to use it in the subsequent algorithms over the large field raised a lot of problems in sage. It turned out that changing the ring of a matrix to the big field was able to solve our issues but it still required going through the code and the scheme and identifying where it was appropriate to change the rings to the big field. Having come from no previous cryptography experience, a lot of new content has been learned throughout the course of this project. In addition, it has required the challenge of understanding fairly advanced mathematical concepts and methods, such as Grobner bases and field lifting.

The functionality of the implementations work well but definitely have room for improvement. For example, during the assignment of random values to the vinegar variables when inverting the core map, if there is no solution to the resulting system the algorithm dictates that we should reassign new random values and try again. In our implementation if there is no solution an error is thrown and then the code is re-run in order to assign a new set of random values to the vinegar variables. Had there been more time this issue could have perhaps been overcome more eloquently.

Administering some form of cryptanalysis to our implementation of the new scheme would have also been valuable had there been more time. Either attacking the scheme ourselves or just implementing existing attacks on the new scheme to provide more of an insight into why they do or do not work would have been useful. Future algebraic cryptanalysis techniques may include practical ways to tackle the MiniRank problem and perhaps more will be known about the complexity of the isomorphism of polynomials problem, amongst others.

8 Conclusion

The main disadvantage regarding multivariate cryptography is the size of the public keys, which are relatively large. Processing larger public keys demand more computational resources and memory which can be restrictive on the wider use of the schemes. The use of field lifting, specifically, taking a public key made over $\mathbb{F}_2$ or some other small field and using it as a public key for a UOV scheme over $\mathbb{F}_{2^r}$ clearly does form smaller public keys [4] since the coefficients of the public key only use one bit, compared to r bits used by each coefficient in a normal UOV scheme over $\mathbb{F}_{2^r}$ [4]. Our implementations show a clear comparison of how the three main algorithms key generation, signature generation and signature verification work in the new scheme with how they work in the normal UOV scheme. The key differences are shown in the implementation of the public key generation algorithm for both of the schemes, we can see from the outputs that the oil vinegar matrices in the new scheme only contain coefficients in $\mathbb{F}_2$, the subsequent transformation T is also made over $\mathbb{F}_2$ and coupling these together we obtain a public key over $\mathbb{F}_2$, our implementation shows that the opposite of this is the case for normal UOV, where we get a public key over $\mathbb{F}_{2^r}$. The new scheme gives us a bit more control over the public key, in that we force the maps F, T and the corresponding pubic key to take values in $\mathbb{F}_2$ which is what we require. The control we have over the size of the public key in the new scheme does not translate into the normal scheme, in the implementation of the normal UOV scheme we allow the core map F to take any random coefficients in $\mathbb{F}_{2^r}$. So once the composition is taken, $P = F \circ T$, the public key takes whatever values the core map F and the random transformation T dictate, this can be seen as a drawback since the public key sizes can get very large if the maps take large values. The results of the implementations are comparable with the details of the two schemes implemented, and if the public keys can be continued to be reduced and schemes such as UOV are continued to be optimised and withstand cryptanalysis, it is entirely possible that multivariate cryptography could be the cryptography of the post quantum era.

References

[1] Jacques Patarin. Hidden fields equations (hfe) and isomorphisms of polynomials (ip): Two new families of asymmetric algorithms. In *International*

Conference on the Theory and Applications of Cryptographic Techniques, pages 33–48. Springer, 1996.

[2] Jacques Patarin. The oil and vinegar signature scheme. In *Presented at the Dagstuhl Workshop on Cryptography September 1997*, 1997.

[3] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 206–222. Springer, 1999.

[4] Ward Beullens and Bart Preneel. Field lifting for smaller uov public keys. In *International Conference in Cryptology in India*, pages 227–246. Springer, 2017.

[5] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[6] Peter W Shor. Polynomial time algorithms for discrete logarithms and factoring on a quantum computer. In *International Algorithmic Number Theory Symposium*, pages 289–289. Springer, 1994.

[7] R Garey Michael and S Johnson David. Computers and intractability: a guide to the theory of np-completeness. *WH Free. Co., San Fr*, pages 90–91, 1979.

[8] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219. ACM, 1996.

[9] Aviad Kipnis and Adi Shamir. Cryptanalysis of the oil and vinegar signature scheme. In *Annual International Cryptology Conference*, pages 257–266. Springer, 1998.

[10] Jacques Patarin, Louis Goubin, and Nicolas Courtois. Improved algorithms for isomorphisms of polynomials. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 184–200. Springer, 1998.

[11] Elwyn R Berlekamp. Factoring polynomials over large finite fields. *Mathematics of computation*, 24(111):713–735, 1970.

[12] Jean-Charles Faugere and Antoine Joux. Algebraic cryptanalysis of hidden field equation (hfe) cryptosystems using gröbner bases. In *Annual International Cryptology Conference*, pages 44–60. Springer, 2003.

[13] Jean-Charles Faugere. A new efficient algorithm for computing gröbner bases (f4). *Journal of pure and applied algebra*, 139(1-3):61–88, 1999.

[14] Jaiberth Porras, John Baena, and Jintai Ding. Zhfe, a new multivariate public key encryption scheme. In *International workshop on post-quantum cryptography*, pages 229–245. Springer, 2014.

[15] Jintai Ding, Jason E Gower, and Dieter S Schmidt. *Multivariate public key cryptosystems*, volume 25. Springer Science & Business Media, 2006.

[16] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In *International Conference on Applied Cryptography and Network Security*, pages 164–175. Springer, 2005.

[17] Jonathan F Buss, Gudmund S Frandsen, and Jeffrey O Shallit. The computational complexity of some problems of linear algebra. *Journal of Computer and System Sciences*, 58(3):572–596, 1999.

[18] Bruno Buchberger. Ein algorithmus zum auffinden der basiselemente des restklassenringes nach einem nulldimensionalen polynomideal. *PhD thesis, Universitat Insbruck*, 1965.

[19] Bruno Buchberger. Bruno buchberger's phd thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. *Journal of symbolic computation*, 41(3-4):475–511, 2006.

[20] Pierre-Jean Spaenlenhauer. *Résolution de systèmes multi-homogènes et determinantiels*. PhD thesis, PhD thesis, Univ. Pierre et Marie Curie-Paris 6, 2012.

[21] Jean-Charles Faugère, Pierrick Gaudry, Louise Huot, and Guénaël Renault. Polynomial systems solving by fast linear algebra, 2013, 2014.

[22] Jean-Charles Faugere, Patrizia Gianni, Daniel Lazard, and Teo Mora. Efficient computation of zero-dimensional gröbner bases by change of ordering. *Journal of Symbolic Computation*, 16(4):329–344, 1993.

[23] Bruno Buchberger. A criterion for detecting unnecessary reductions in the construction of gröbner-bases. In *Symbolic and algebraic computation*, pages 3–21. Springer, 1979.

[24] David Eisenbud. *Commutative Algebra: with a view toward algebraic geometry*, volume 150. Springer Science & Business Media, 2013.

[25] Daniel Lazard. Gröbner bases, gaussian elimination and resolution of systems of algebraic equations. In *European Conference on Computer Algebra*, pages 146–156. Springer, 1983.

[26] Magali Bardet. Asymptotic expansion of the degree of regularity for semi-regular systems of equations. *MEGA'05*, 2005.

[27] Jean-Charles Faugere. A new efficient algorithm for computing gröbner bases without reduction to zero (f5)(15/6/2004).

[28] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 392–407. Springer, 2000.

[29] Nicolas T Courtois and Jacques Patarin. About the xl algorithm over gf (2). In *Cryptographers' Track at the RSA Conference*, pages 141–157. Springer, 2003.

[30] Weiwei Cao, Lei Hu, Jintai Ding, and Zhijun Yin. Kipnis-shamir attack on unbalanced oil-vinegar scheme. In *International Conference on Information Security Practice and Experience*, pages 168–180. Springer, 2011.

[31] Nicolas Courtois and CP Bull. The minrank problem, 2000.

[32] Aviad Kipnis and Adi Shamir. Cryptanalysis of the hfe public key cryptosystem by relinearization. In *Annual International Cryptology Conference*, pages 19–30. Springer, 1999.

A Appendix

All code is done in SageMath. In the .zip there are 2 files, one called "Lifted UOV.sagews" and another called "UOV - field characteristic 2.sagews". In order to run the code please do the following for each of the files:

Step 1: Open the file "UOV - field characteristic 2.sagews". Open the file in a source code editor that will preserve the format of the sage code, such as gedit or notepad++.

Step 2: Go to the online SageMath editor called CoCalc at https://cocalc.com/ and create an account there as this is where the code was written.

Step 3: Create a new project in CoCalc and then once in the new project click "Create or upload files..."

Step 4: Now either create a new file and select sage worksheet and then copy and paste the code from the file "UOV - field characteristic 2.sagews" into the blank sage worksheet on CoCalc. Or simply drag and drop the file into the section where it says "upload files from your computer" and the sage worksheet should be added to the project.

Step 4: If you have copied and pasted the code into a blank sage worksheet then run the code by pressing the green run bottom at the top, ensuring the text cursor is at the beginning of the first line of code in the file. If you have uploaded the file then navigate to "files" located in the top left of the screen, this will take you to the files in the project and then click on the file and the sage worksheet should open, and then run this code in the same way that has been described above. In each case the output of the code will be displayed at the bottom of each sage worksheet.

Step 5: If any errors occur after pressing run simply press run again.

Now do all of the above again for the second file called "Lifted UOV.sagews". **Note** if the files are being uploaded into the project on CoCalc instead of copying and pasting the code into a blank sage worksheet. Then it is easier to drag and drop both files into the project at once and then run each file separately. Once in one file and finished running it, click the files button in the top left of the screen to go back to the files in the project and then you can open and run the second file.

The **regular UOV** scheme is the file titled "UOV - field characteristic 2.sagews" and the **new scheme with field lifting** is the file called "Lifted UOV.sagews". In each of the files you can alter the parameters of o and q to see the effects but the field is of characteristic 2 in each file, so q must be of the form 2^r.

```
Generating Oil-Vinegar matrices...
[1 1 0 0|0 0]
[0 1 0 1|0 1]
[1 1 1 0|0 0]
[0 0 0 0|1 1]
[-------+---]
[0 0 1 1|0 0]
[0 1 0 0|0 0]

[x0^2 + x0*x1 + x1^2 + x0*x2 + x1*x2 + x2^2 + x1*x3 + x2*x4 + x3*x5]

[1 1 1 0|0 1]
[1 1 0 0|0 1]
[0 0 1 0|0 1]
[0 0 0 1|1 1]
[-------+---]
[1 1 1 0|0 0]
[0 0 0 0|0 0]

[x0^2 + x1^2 + x0*x2 + x2^2 + x3^2 + x0*x4 + x1*x4 + x2*x4 + x3*x4 + x0*x5 + x1*x5 + x2*x5 + x3*x5]

Generating public key polynomials...
[x0*x1 + x0*x2 + x2^2 + x1*x3 + x3^2 + x0*x4 + x2*x4 + x4^2 + x0*x5 + x2*x5 + x4*x5]
[x0^2 + x0*x1 + x0*x2 + x1*x2 + x1*x3 + x2*x4 + x3*x4 + x0*x5 + x3*x5 + x4*x5 + x5^2]

Core map F...
[x0^2 + x0*x1 + x1^2 + x0*x2 + x1*x2 + x2^2 + x1*x3 + x2*x4 + x3*x5]
[x0^2 + x1^2 + x0*x2 + x2^2 + x3^2 + x0*x4 + x1*x4 + x2*x4 + x3*x4 + x0*x5 + x1*x5 + x2*x5 + x3*x5]

Hash result:
[10, 6]

Solving for oil variables...
{x5: a + 1, x4: a^4 + a^3 + a^2 + a + 1}

Pre-image of the hash value under the core map F...
[a^2, a^4 + a^2, a^4 + a^2, a^4 + a^2 + a + 1, a^4 + a^3 + a^2 + a + 1, a + 1]

Computing signature...
(a^3 + a^2 + a + 1, a^4 + a^2, a^4 + a^2 + a + 1, a^3 + a + 1, a^4 + a^2, a^4 + a^3 + a + 1)

Signature validation...
True
```

Figure 1: Random output of the implementation of the lifted UOV scheme [4]. The following parameters where used: $q = 2^5, o = 2, v = 2o, n = o + v, K.\langle a \rangle = GF(q), K2.\langle a \rangle = GF(2)$

```
Generating Oil-Vinegar matrices...
[               1          a^2 + 1          a^2 + 1          a^3 + a|          a^3 + a          a^3 + a]
[      a^3 + a + 1          a^3 + a              a^2        a^3 + a^2|          a^3 + 1     a^3 + a^2 + 1]
[        a^3 + a^2          a^2 + 1          a^3 + 1                a|a^3 + a^2 + a + 1              a + 1]
[             a^2              a^3      a^3 + a + 1    a^3 + a^2 + 1|          a^2 + a                a^2]
[-------------------------------------------------------------------+-----------------------------------]
[          a^3 + a     a^3 + a^2 + a          a^2 + a                a|                0                 0]
[               1         a^3 + a^2              a^3            a + 1|                0                 0]

[x0^2 + (a^3 + a^2 + a)*x0*x1 + (a^3 + a)*x1^2 + (a^3 + 1)*x0*x2 + x1*x2 + (a^3 + 1)*x2^2 + (a^3 + a^2 + a)*x0*x3 +
(a^2)*x1*x3 + (a^3 + 1)*x2*x3 + (a^3 + a^2 + 1)*x3^2 + (a^2 + a + 1)*x1*x4 + (a^3 + 1)*x2*x4 + (a^2)*x3*x4 + (a^3 + a +
1)*x0*x5 + x1*x5 + (a^3 + a + 1)*x2*x5 + (a^2 + a + 1)*x3*x5]

[    a^3 + a^2 + 1          a^3 + a     a^3 + a^2 + a              a^3|            a + 1       a^3 + a + 1]
[    a^3 + a^2 + a                a                1          a^2 + a|                1                a^2]
[               1                1     a^3 + a^2 + a          a^2 + a|          a^2 + 1          a^3 + 1]
[a^3 + a^2 + a + 1                1     a^3 + a^2 + a          a^3 + a|                1                a^2]
[-------------------------------------------------------------------+-----------------------------------]
[      a^2 + a + 1       a^2 + a + 1              a^3              a^2|                0                 0]
[               1          a^3 + a       a^2 + a + 1       a^3 + a + 1|                0                 0]

[(a^3 + a^2 + 1)*x0^2 + (a^2)*x0*x1 + (a)*x1^2 + (a^3 + a^2 + a + 1)*x0*x2 + (a^3 + a^2 + a)*x2^2 + (a^2 + a + 1)*x0*x3 +
(a^2 + a + 1)*x1*x3 + (a^3)*x2*x3 + (a^3 + a)*x3^2 + (a^2)*x0*x4 + (a^2 + a)*x1*x4 + (a^3 + a^2 + 1)*x2*x4 + (a^2 + 1)*x3*x4
+ (a^3 + a)*x0*x5 + (a^3 + a^2 + a)*x1*x5 + (a^3 + a^2 + a)*x2*x5 + (a^3 + a^2 + a + 1)*x3*x5]

Generating public key polynomials...
[(a^3)*x0^2 + (a^3 + 1)*x0*x1 + (a^3)*x0*x2 + (a^3 + a^2 + 1)*x1*x2 + (a^2 + a + 1)*x2^2 + (a^3 + a)*x0*x3 + (a^3 + a^2 + a +
1)*x1*x3 + (a^3 + a^2 + a + 1)*x3^2 + (a^2 + 1)*x0*x4 + (a^3 + 1)*x1*x4 + (a^3 + a + 1)*x2*x4 + (a^3 + a^2)*x4^2 + (a^3 + a^2
+ 1)*x0*x5 + (a^3 + a)*x2*x5 + (a^2)*x3*x5 + (a^3 + a^2 + a)*x4*x5 + (a^3 + a^2 + a)*x5^2]
[(a + 1)*x0*x1 + (a^3 + a^2 + 1)*x1^2 + x0*x2 + (a^3 + 1)*x1*x2 + (a^3 + a)*x2^2 + (a^3 + a + 1)*x0*x3 + (a^2 + a)*x1*x3 +
(a^2 + 1)*x2*x3 + (a^3 + a^2 + 1)*x3^2 + x0*x4 + (a^2 + 1)*x1*x4 + (a + 1)*x2*x4 + (a^3 + a^2 + a)*x3*x4 + (a^3 + 1)*x4^2 +
x0*x5 + (a^3)*x3*x5 + (a + 1)*x4*x5 + (a^2 + 1)*x5^2]

Core map F...
[x0^2 + (a^3 + a^2 + a)*x0*x1 + (a^3 + a)*x1^2 + (a^3 + 1)*x0*x2 + x1*x2 + (a^3 + 1)*x2^2 + (a^3 + a^2 + a)*x0*x3 +
(a^2)*x1*x3 + (a^3 + 1)*x2*x3 + (a^3 + a^2 + 1)*x3^2 + (a^2 + a + 1)*x1*x4 + (a^3 + 1)*x2*x4 + (a^2)*x3*x4 + (a^3 + a +
1)*x0*x5 + x1*x5 + (a^3 + a + 1)*x2*x5 + (a^2 + a + 1)*x3*x5]
[(a^3 + a^2 + 1)*x0^2 + (a^2)*x0*x1 + (a)*x1^2 + (a^3 + a^2 + a + 1)*x0*x2 + (a^3 + a^2 + a)*x2^2 + (a^2 + a + 1)*x0*x3 +
(a^2 + a + 1)*x1*x3 + (a^3)*x2*x3 + (a^3 + a)*x3^2 + (a^2)*x0*x4 + (a^2 + a)*x1*x4 + (a^3 + a^2 + 1)*x2*x4 + (a^2 + 1)*x3*x4
+ (a^3 + a)*x0*x5 + (a^3 + a^2 + a)*x1*x5 + (a^3 + a^2 + a)*x2*x5 + (a^3 + a^2 + a + 1)*x3*x5]

Hash result:
[2, 10]

Solving for oil variables...
{x5: a^3 + a^2 + 1, x4: a + 1}

Pre-image of the hash value under the core map F...
[a^3 + 1, a^2 + a + 1, a^3, a^3 + a + 1, a + 1, a^3 + a^2 + 1]

Computing signature...
(a^2 + 1, a^3 + a + 1, a^3 + a^2 + a, a^2 + 1, a^3 + a^2 + a + 1, a^2)

[0, 0]
Signature validation...
True
```

Figure 2: Random output of the implementation of the normal UOV scheme [3]. The following parameters where used: $q = 2^4, o = 2, v = 2o, n = o + v, K.\langle a \rangle = GF(q)$