# How To Generate $E_2$ Polynomials using SageMath

Nicolas T. Courtois[1], Peter Spacek[2], and Students[1]

[1] Computer Science department, University College London, UK
[2] Slovak University of Technology

**Abstract.** Based on earlier ideas by Matsumoto-Imai and HFE a new post-quantum public key cryptosystem called Two-Face was proposed in 2017 cf. [13]. A paper will appear in Africacrypt 2018. TBC.
In this paper, we study the choice of an internal hidden univariate polynomial in all these cryptosystems. We show how to implement and test the security of these cryptosystems with SageMath software. We show that many interesting choices have low regularity degree and therefore are insecure. We present some new candidates which have higher regularity degree and therefore seem more secure.

**Key Words:** applied cryptography, digital signatures, post-quantum crypto, HFE, Matsumoto-Imai, TwoFace, permutation polynomials

**Note.** This paper is work in progress and was written as a tutorial for UCL students doing GA18 Cryptanalysis project PQ.

## 1 Multivariate Cryptography

### 1.1 Matsumoto-Imai vs. RSA

The Matsumoto-Imai public key cryptosystem, also known as $C^*$ or MI, is one quite old public key cryptosystem proposal first introduced at Eurocrypt'88 [7, 5, 8]. It is based on a hidden univariate polynomial transformation, which can be for example $x \mapsto x^3$. Or more generally on a power of the form $x \mapsto x^{q^k+q^l}$ for finite fields of type $GF(q^n)$ where $q$ is a small power of a prime. A popular choice is $q = 2$ or $q = 256$.
Let us compare MI to RSA. In both cases, we can have the same cube function

$$E_1 : x \mapsto x^3.$$

However in MI instead of being over a ring of numbers modulo some $N$ like with RSA, is over a finite field, for example, $GF(2^{80})$. The order of the multiplicative group of $GF(2^{80})$ is known and therefore in many cases, such a power function over a finite field is, unlike in RSA, easily invertible [and the inverse is typically also another power function].
In RSA this univariate polynomial is known to the attacker and the factors are secret. In MI can be further "obfuscated" and does no longer resemble a univariate polynomial. In MI the polynomial will be "concealed" (cf. [7, 8]) and transformed into a set of multivariate quadratic polynomial functions $GF(q)^n \to GF(q)^n$. There are two additional $GF(q)$-affine transformations $S$ and $T$ in $GF(q)^n \to GF(q)^n$ and we consider a composition:

$$F_1 = T \circ E_1' \circ S$$

where $E_1'$ is the original polynomial $E_1$ transformed into $n$ quadratic equations with $n$ variables over $GF(q)$ and $F_1$ will become the public key. Thus, Matsumoto and Imai construct their public key cryptosystem, see [7, 8, 5] for more details.

### 1.2 Cryptanalysis of Matsumoto-Imai

At Crypto'95, Jacques Patarin presents an attack on MI. He shows that there exist a bi-linear or rather a bi-affine set of polynomial equations

$$F_2(x_1, x_2, \ldots ; y_1, y_2, \ldots)$$

such that if we substitute $y$ we get a system which is linear in the $x_i$. Thus the cryptosystem is broken, see [5, 8]. With conventions used in [3, 2] we say that MI admit equations of type $XY$. We would call type $xy$ bi-linear equations and type $XY$ bi-affine equations. From here there are two major ways to break MI:

A We recover the equations $F_2()$ (by interpolation and linear algebra) and MI can be broken by substitution of $y$ in $F_2()$ and solving linear equations! A is an indirect and not very popular method, with precomputation. This is how MI was broken for the first time [7] and how HFE was first broken [2].

B Do computations in the ideal generated by equations $F_1()$ directly. Once the output $y$ is known we can multiply these equations by variables and obtain new equations of degree 3. Then due to the existence of $F_2()$, but without computing $F_2()$ directly, so-called **degree falls** are obtained: some new polynomials of degree 2 are generated. We repeat this process and the number of linearly independent polynomials of degree 2 grows and grows, and, eventually, the degree falls to 1, linear polynomials are also generated. B is a direct and more popular method which has no pre-computation and numerous software implementations.

Typically what happens in method $B$ is that never during the whole computation the degree will exceed 3. In modern algebraic cryptanalysis vocabulary, we say that $F_1() = y$ has a degree of regularity 3. In rare cases, the process could stop [or rather generate only the same equations again] and it could be necessary to generate polynomials of higher degree $> 3$.

**Cost**. In general Method A is cheaper than Method B if we ignore the cost of pre-computing the relations $F_2(x_1...y_1...)$. Method A also reveals that many computations traditionally done during algebraic attacks of type B are redundant or subject to pre-computation(!). Then routine code breaking with different $y$ can be done routinely at a lower cost. Most existing crypto literature completely ignores this problem and therefore the security of numerous public key and digital signature schemes has not been evaluated accurately or not yet, as only method B is typically studied.

## 1.3 Regularity Degree and Degree Falls

This term is an object of some confusion in the literature. In general, mathematicians define "regular" sequences of polynomials, a theoretical property which is violated for most real-life sets of polynomials over finite fields [1]. From this authors define a "semi-regularity degree" [1] which is frequently equal to a maximum degree reached during the Gröbner basis computation.

**Definition 11 (Informal - Regularity Degree).** In crypto literature authors frequently use the term "regularity degree" to denote the degree of the first-degree fall in Gröbner basis computations with algorithms such as F4 or F5, ignoring the fact that in some rare cases this degree further increases during towards the end of the computation [which is rare] and that this degree may in some rare cases be different for different algorithms [F5 vs F4 etc].

## 1.4 Second Face - Multivariate Version

We can say that $F_2()$ is **a second face** of $F_1()$. However $F_1()$ is a function, or it has an "explicit" form, we compute $y$ from $x$ explicitly. In contrast $F_2(x, y)$ are I/O relations, (Input/Output relations) or it has an "implicit" form.

## 1.5 Second Face - Two-Variate Versions $E_2$

The second face also always or frequently occurs at the level of polynomials with “big” variables $x, y$. The only problem is sometimes that the degree of these polynomials is very large or they have too many terms.

**Simple and Old Example 1**. For example from [8] we learn that if

$$y = E_1(x) = x^3,$$

then we have

$$E_2(x, y) = xy - x^4.$$

Here when $y$ is fixed the univariate degree in $x$ becomes 4 and a multivariate degree in $x_i$ becomes 1. Breaking this cryptosystem requires time of about

$$\mathcal{O}(n^{d\omega})$$

where $\omega \le 2.39$ it the exponent of the Gaussian reduction This is at most $\mathcal{O}(n^3)$ time.
The degree of regularity is 3 here, and method B is less efficient than method A with complexity at most $\mathcal{O}(n^{3\omega})$.

**More General MI Attack - Old Example 2**. In general from [8] we learn that if

$$y = E_1(x) = x^{q^a+1},$$

then we have the identity $q^{2a} + (q^a + 1) = q^a(q^a + 1) + 1$ and therefore we have

$$E_2(x, y) = x^{q^{2a}}y - y^{q^a}x.$$

Degree with method 1 is still just 1 here and attacks take less than $\mathcal{O}(n^3)$ time. Method B will again operate at degree 3 in time at most $\mathcal{O}(n^{3\omega})$.

**HFE Example 3**. We can have for example the following polynomial which is a special case of the HFE cryptosystem [9]:

$$y = E_1(x) = x + x^3 + x^5,$$

then we have

$$E_2(x, y) = ????$$

which can be computed following works of Joux and Faugère, and will NOT in general have low degree in $x$, however $F_2(x_1, x_2, \ldots ; y_1, y_2, \ldots)$ will have low degree in $x_i$ and the degree of regularity will be lower. It is possible to see that if $D$ is degree of $E_1(x)$ then the degree of regularity of $F_2$ will increase as $\log_2(D)$, [2]. This was further studied by Joux and Faugère and numerous other authors (more recently by Kosters [6]).

**New Example 4**. In [13] we have the following example based on a Dobbertin permutation polynomial: Let $n = 2m - 1$.

$$y = E_1(x) = x^{2^m+1} + x^3 + x,$$

then we have

$$E_2(x, y) = x^9 + x^6 y + x^5 + x^4 y + x^3(y^{2^m} + y^2) + xy^2 + y^3 = 0$$

We do no longer have $E_1$ of low degree. Instead we get a **low degree in x** for $E_2$. When $y$ is fixed the univariate degree in $x$ becomes $D = 9$. This is useful for private key operations such as signature generation: Polynomial equations of degree 9 can be easily solved in finite fields. Now the multivariate degree in $x_i$ becomes 2 (for example we have $9 = 2^3+1$). the complexity to break this cryptosystem by method A ignoring precomputation of relations is about $\mathcal{O}(n^{2\omega})$.
The degree of regularity is probably at most 4 here. For the attacker who does not know $S, T, E_2()$ but only $F_2$, the complexity to break this cryptosystem by method B is higher and about $\mathcal{O}(n^{4\omega})$.

### 1.6 Crypto Design Goals

Let $D$ be the univariate degree of $E_2$ in $x$ and when $y$ is fixed.
The designer tries to increase the degree or regularity $d$ for $F_2$ while keeping the univariate degree $D$ of $E_2$ relatively low.

### 1.7 Security vs. Two-Variate Versions

The goal of security evaluations is to see that we do not have another "better" $E_2$ which exists. An ultimate test is to evaluate the degree of regularity of the multivariate version $F_2(x) = y$ for some fixed $y$ value: a real-life attack with a Gröbner bases algorithm. The degree of regularity is the same for $F_1$ and $F_2$.

## 2 Computing $E_2$ with Resultants

In [13] we find an explicit formula to compute $E_2$ using resultants. This can be implemented in SageMath.
We go back to New Example 4 from [13].

$$y = E_1(x) = x^{2^m+1} + x^3 + x,$$

Let us recode this as

$$y = B(x,z) = xz + x^3 + x$$

where $z = x^{2^m}$ is the high degree part.
A nice trick to compute $E_2$ is to use resultants as explained in [13]. We observe that if $z = x^{2^m}$ we also have $x = z^{2^{m-1}}$ as $n = 2m-1$.
Therefore one way to rewrite $y = B(x,z)$ is $y = B(z^{2^{m-1}}, x^{2^m})$.
Then we compute:

$$E_2(x,y) = Res_z\left(B(x,z) - y, \quad \left(B(z^{2^{m-1}}, x^{2^m})\right)^{2^m} - y^{2^m}\right)$$

Here is our SageMath code in which we manually replace things to avoid any high powers to appear. For example, we observe that $n = 2m-1$ and $z^n$ is the same as $z$ and that

$$(z^{q^{m-1}})^{q^m} = z^{q^{m-1}*q^m} = z^{q^m*q^m*q^{-1}} = z^{q^{2m-1}} = z^{q^n} = z^1 = z$$

and

$$(x^{q^m})^{q^m} = x^{q^m*q^m} = x^{q^{2m*1}} = x^{q^{2m}*q^{-1}*q^1} = x^{q^{2m-1}*q} = x^{q^n*q} = x^{1*q} = x^q$$

So the rule is that $x$ can be replaced by $z$, and $z$ must be replaced by $x^2$. In this case we need to write:

```
P.<x,y,z,t> = PolynomialRing(GF(2^51), 4)
B1 = x*z + x^3 + x - y
B2 = z*x^2 + z^3 + z - t
B1.resultant(B2, z)
```

It is possible to see that the result is the same for any sufficiently larger finite field of characteristic 2.
The result is:

$$x^9 + x^6 * y + x^5 + x^4 * y + x^3 * y^2 + x^3 * t + x * y^2 + y^3$$

where $t$ can now be replaced by $y^{2^m}$.
It is important to see that $E_2$ is NOT at all unique, many such polynomials exist. However, the resultant method above is guaranteed to give polynomials with a degree in $x$ which is not too high, being at most $qd^2$ where $q = 2, d = 3$ here, see [13]. Therefore at most 18. Yet **better** polynomials with a lower degree in $x$ may exist. By our SageMath method, we obtained a lower degree 9.

### 2.1 Simple Pat Example 5

We apply the same SageMath code to SimplePat Example 1 on page 6 of [13].

```
P.<x,y,z,t> = PolynomialRing(GF(2^51), 4)
B1 = x*z + x^5 + x^3 - y
B2 = z*x^2 + z^5 + z^3 - t
B1.resultant(B2, z)
```

The computation takes 10 seconds and we get

$$x^{25}+x^{23}+x^{20}*y+x^{13}+x^{9}+x^{8}*y+x^{7}*y^{2}+x^{5}*y^{4}+x^{6}*y+x^{5}*y^{2}+x^{3}*y^{4}+x^{5}*t+x^{2}*y^{3}+y^{5}$$

which is the same as in the paper.

### 2.2 General Pat Example 6

Again the rule is that $x$ can be replaced by $z$, and $z$ must be replaced by $x^2$. In this case we need to write:

```
P.<x,y,z,t> = PolynomialRing(GF(2^57), 4)
B1 = x^3 + x*z + z^3 - y
B2 = z^3 + x^2*z + x^6 - t
B1.resultant(B2, z)
```

We get the same result as in the paper.

## 3 Software generated polynomials and $E_2$

Here we present an automated way to generate more polynomials in each family, and computing $E_2$:

### 3.1 Simple Pat

In Simple Pat family of polynomials have following form:

$$E_1(x) = P(x) = x^{q^{m+1}} + \sum_{i=0, i=q^j, i=q^j+q^k}^{i \leq d} \alpha_i x^i$$

Mohammed Mirza, Jack Rowland and Maxine Emuobosa wrote a code for generating polynomials in Simple Pat Family. You can use following code to generate all $E_1$ polynomials up to certain degree, along with $E_2$ computation:

```
from sets import Set
iSet = Set([0])
q = 2; d = 5
j = 0; k = 0

while True:
    k = 0
    while True:
        i = (q**j, q**j + q**k)
        if i[1] > d:
            break
        iSet.add(i[0])
        iSet.add(i[1])
        k += 1
    if i[0] > d:
        break
    j += 1

powers = list(powerset(iSet))
powers.pop(0)
print powers

P.<x,y,z,t> = PolynomialRing(GF(2^51, 'x'), 4)
B1list = map(lambda p: x*z + sum(map(lambda y: x^y, p)) - y, powers)
B2list = map(lambda p: z*x^2 + sum(map(lambda y: z^y, p)) - t, powers)

for i in range(len(B1list)):
    print (B1list[i] + y, B1list[i].resultant(B2list[i], z))
```

### 3.2 General Pat

For generating polynomials which follows the rule

$$B(x,z) = \sum_{i=0, i=q^j, i=q^j+q^k}^{i\le d} \alpha_i x^i + \sum_{i=q^j, i=q^j+q^k}^{i\le d} \beta_i z^i + \sum_{i=q^k, j=q^l}^{i+j\le d} \gamma_{i,j} x^i z^j,$$

Marios Georgiou and Alex Nikas, created SageMath function. The SageMath code for B_xz function is in the appendix. You can use it with this command:

```
e1 = B_xz(2, 3, x, z, [0, 0, 0, 1], [0, 0, 1], [1, 0, 0])
print e1
```

First two arguments are $q$ and $d$. Next two arguments are variables $x$ and $z$. Three vectors at the end are $\alpha$, $\beta$ and $\gamma$. $\alpha_i$ is then i th element of alpha (alpha[i]). We present code for computing $E_2$. WRT the paper we let $t = y^{q^m}$, $z = x^{q^m}$. Alpha, beta and gamma are again coeficients for $E_1$

```
def E_2(q, d, x, y, z,t, alpha, beta, gamma):
    b1 = B_xz(q, d, x, z, alpha, beta, gamma)
    b2 = B_xz(q, d, z, x^q, alpha, beta, gamma)
    res1 = b1 - y
    res2 = b2 - t
    E2 = res1.resultant(res2, z)
    return E2
```

To generate $E_2$ type following:

```
e2 = E_2(2, 3, x, y, z, t, [0, 0, 0, 1], [0, 0, 1], [1, 0, 0])
print e2
```

### 3.3 MAC - Permutation Polynomial Generation

**Settings:** The field size is specified by variable $q$ presented in form $q = 2^p$, where $p$ is positive integer. The degree of the field extension is $n$ and has to be in the form $2m - 1$.

**Generation:** To generate our Permutation Polynomials we sieved general PAC $E_1$ polynomials that were randomly generated by group three's general PAC generator using Hermite's Criterion [15]. This first required us to convert the $E_1$ polynomials from a multivariate polynomial in terms of $x$ and $z$ to a univariate polynomial $E_{1,uni}$ in the terms of $x$, this was done by substituting $z$ with $x^{q^m}$ as shown in two face paper [13]. This univariate polynomial must have exactly one root in the field $F_{q^n}$ and

$E_{1,uni}(x)^t\ mod\ (x^q - x)$ must have a degree $< q-2$ for every positive integer $t$ where $t \leq q-2$ and $t\ mod\ p \neq 0$. When these conditions hold $E_{1,uni}$ is a Permutation polynomial for the chosen Settings and a valid $E_2$ can be generated.

**Contribution:** The work was equally split between the Simon Bohm, Quentin Delmas and Bryce Boyd, with several iterations created and merged into the final generator. As mentioned above the $E_1$ general PAC generator was created by group three. The MAC $E_2$ generator which is passed valid $E_1$ polynomials was created by all five members of group one. There is also an incomplete security evaluation function included with the Generator.

**Example Polynomials:**
- p: 1 d: 4 m: 3 q: 2 n: 5
  E1: $x^{32} + x^9 + x^8$
  E2: $x^{32} + y^{32} + x^{15} + x^7y^8 + x^{14} + x^6y^8 + x^9 + x^8y$
- p: 1 d: 6 m: 5 q: 2 n: 9
  E1: $x^{160} + x^{96} + x^{32}$
  E2: $y^{160} + x^{10}y^{128} + x^6y^{128} + x^2y^{128} + x^{40}y^{32} + x^{24}y^{32} + x^{50} + x^{46} + x^{42} + x^8y^{32} + x^{34} + x^{30} + x^{26} + x^{18} + x^{14} + x^{10}$
- p: 1 d: 6 m: 5 q: 2 n: 9
  E1: $x^{66}$
  E2: $y^{64} + x^4y^2$
- p: 1 d: 4 m: 5 q: 2 n: 9
  E1: $x^{128} + x^{64} + x^4 + x^2 + x + 1$
  E2: $y^{128} + y^{64} + x^{32} + x^{16} + x^8 + x^4 + y^4 + y^2 + x + y + 1$
- p: 2 d: 6 m: 3 q: 4 n: 5
  E1: $x^{64} + x^4 + x^2$
  E2: $x^{16} + y^8 + x^4 + y^4 + x^2 + y^2$
- p: 2 d: 4 m: 3 q: 4 n: 5
  E1: $x^{256} + x^{128} + x^4$
  E2: $x^{32} + y^{32} + x^{16} + y^{16} + x^8 + y^4$

**Source code:** The source code for generation can be found in appendix.

### 3.4 Three or a few more Blocks, 'Super Two-Face'

Maxine and Jack tried to solve a question of Super Two-Face. The first try was to use Macaulay Resultant to compute resultant of three polynomials. This was a dead-end because for the Macaulay Resultant we need three polynomials of variables, and these polynomials have to be homogenous. But this condition is not possible for $B_1$,$B_2$ and $B_3$:

$$B(x, z_1, z_2) = y$$

$$B(z_1^{q^{2m-1}}, z_2^{q^{2m-1}}, x^{q^{2m}})^{q^m} = y^{q^m}$$

$$B(z_2^{q^{2m-1}}, x^{q^m}, z_1^{q^2})^{q^2m} = y^{q^2m}$$

After realizing this we found a way of computing $E_2$ from General Pad polynomial:

$$E_1(x) = x^2 + x * z_1 + x * z_2 + x + 1$$

Computation may be done with this code provided by Maxine:

```
q=2
# t1= y^(q^m), t2=t1^(q^m)
var('t1, t2, x, z1, z2, m, y')
E1 = y == x^2 + x*z1 + x*z2 + x + 1
E1qm = E1.subs(y=t1,x=z1,z1=z2,z2=x^q)
# subs here is subbing values such that E1qm = E1^(q^m)
E1qm2 = E1qm.subs(t1=t2,x=z1,z1=z2,z2=x^q)
# subs here is subbing values such that E1qm2 = (E1^(q^m))^(q^m)
print E1; print E1qm; print E1qm2; print

eqnz2 = solve(E1,z2)
# solving equation E1 for z2.
print eqnz2[0]
az2 = eqnz2[0].right()
ieqn = E1qm.subs(z2=az2)
# subbing z2 for its value in terms of x, y and z1
print "Intermediate eqn: "+ str(ieqn); print
# Intermediate equation is E1qm with z2 subbed out

eqnz1 = solve(ieqn, z1)
# subbing z1 for its value in terms of x, y and t1
print eqnz1[0]
az1 = eqnz1[0].right()
if(eqnz1[0].left()==0):
    raise RuntimeError, "No roots found"
print

peqn = E1qm2.subs(z2=az2).subs(z1=az1)
# peqn or 'prenultimate' equation is E1qm2
# in terms of x, y, t1 and t2 with z1 and z2 subbed out.
E2= peqn.right()-t2
print "E2: "+ str(E2)
```

This can be done for many more polynomials $E_1$. in General Pad form.

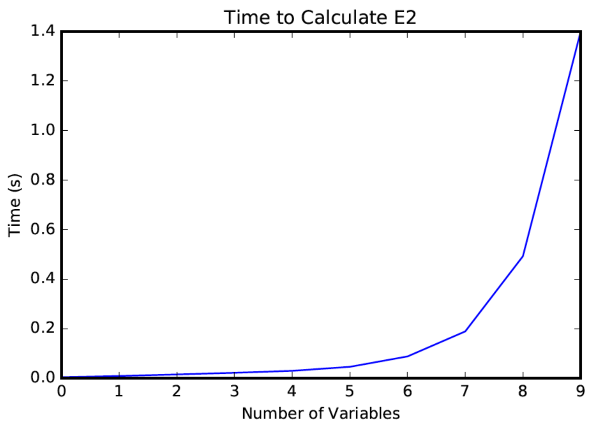
## 3.5 More Block

Michael Suinn decided to make his own multivariate polynomial class so that I can store, manipulate, and evaluate the polynomials being generated in the General Pat algorithm for any number of input variables. The class is very straightforward with the standard iterators, setter functions, and one getter function. The evaluate function is very straightforward as it evaluates the stored polynomial with the given arguments. There was one thing that is technically broken, the symbolic evaluation function as it is unable to do evaluate multi-term expressions, but the algorithm never uses them so Michael felt that the current state is fine. Michael thinks the print function is pretty self-explanatory.

Michael tried to tackle the task of creating E2 for arbitrary numbers of variables in E1 by writing a multivariable polynomial class and B generator function to store and manipulate polynomials. It can generate B and E1, but E2 becomes computationally intractable due to the exponential increase in the size of the Sylvester matrix used to calculate the resultant.

Timing results from the code clearly confirm Michael's exponential complexity growth conclusion. For  15-20 variables, SageMath overflows and is unable to calculate the resultant. Michael uses successively larger powers of x in b2 because of the way that the variables are defined in the paper ($z_1 = x^q, z_2 = z1^q...$) and assume the absolute best case scenario, where all but the largest power of x is canceled out. Even then, at  10 variables, it takes 1+ seconds to calculate E2. This conclusion is also indirectly supported by the paper, stating that it is an open problem to find B such that E2 is easy to calculate, implying that the general B is difficult. The graph for 10 variables is included below:

**Fig. 1.** Time dependency diagram for calculating E2



## 4 How to generate public key

We need to generate public keys from the $E_1$ We will use construction described in [5]. Let $x$ be column vector $x = (x_1, x_2, ..., x_n)$ and $y$ analogicly $y = (y_1, y_2, ..., y_n)$. We generate two random $n$ x $n$ affine matrices S and T (T has to be invertible), and two random vectors $v_s$ and $v_t$. We compute two vectors $u$ and $v$:

$$u = (S * x) + v_s$$

and

$$v = (T * y) + v_t$$

More pricesly we want to compute $y$ from this equation.

$$y = T^{-1} * (v - v_t)$$

The relation between $u$ and $v$ is given by equation

$$v = E_1(u)$$

We attach SageMath code for each family of polynomials in the appendix.

## 5 Implementation of Two-Faces signature scheme

We decided to implement Signature and Verification of Two-Faces multivariate system. The code produced by Cindy Hau and Tereza Loffelmannova is in the appendix. This code is not finished and should be rewritten in the future.

```
Data: message $x$ of length $n$, secret key(matrices $S$ and $T$, vectors $v_s$ and $v_t$),
      polynomial $E_2$
Result: signature $sig$ (list of polynomials)
1. Compute $u$ using the equation: $u = Sx + v_s$
$u$ is in the form of vector of zeroes and ones.
2. Compute $v$ using the equation: $v = Ty + v_t$
$v$ is in the form of vector of polynomials with $y_i$
3. Compute $sig = E_2(u, v)$.
$sig(y)$ is in form of list of polynomials with $y_i$
#note $sig(y) = (00..0)$ only if $y = Pub(x)$, public key was generated with $E_1$
4. return $sig$
```

**Algorithm 1:** Signature pseudoalgorithm

```
Data: message $x$ of length $n$, public key(vector of polynomials $y$, signature $sig$
      (list of polynomials)
Result: true or false
1. Compute $y = Pub(x)$
$y$ is in the form of vector of zeroes and ones.
2. Compute $sigRes = sig(y)$
if $sigRes$==0 then
    return true
else
    return false
end
```

**Algorithm 2:** Verifycation pseudoalgorithm

## 6 Verification

$E_2$ is not unique, therefore it is useful to write software which checks if polynomials with a yet lower degree in $x$ do not exist (while remaining sparse and keeping degree in $y$ not too high). This is difficult due to sparsity conditions.

## 7 Security Simulations

For each $E_1$ we need to check if the regularity degree is high while minimizing the degree in $x$. This will give best cryptosystems for industrial applications.

Now, with $y$ equations from section 4, we evaluate security with method B. We use SageMath for computing ideal and also computing degree of semi regularity and actual Groebner basis. We need to add field equations to the computation of ideal:

$$x_i^2 = x_i \forall i, 0 \leq i < n$$

```
L = []
for f in y:
    L.append(P(f[0]).homogenize())

for i in (0..n-1):
    L.append(P(var('x%i' % i)^2-var('x%i' % i)).homogenize())

I = Ideal(L)

print I.degree_of_semi_regularity()
print max(f.degree() for f in I.groebner_basis())
```

If we don't homogenize the polynomials, we have complete results. We are not able to explain it now.

We present more polynomials with security evaluation:

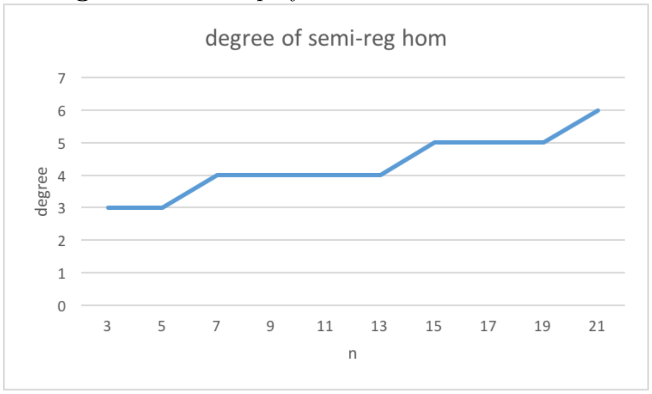
### 7.1 Simple Pat

Here we present result of student's polynomials tests. First, Maxine tested polynomial $x^3 + x^2 + x * z + x$:

**Table 1.** Results for polynomial: $x^3 + x^2 + x * z + x$

| n | deg s-reg | max deg in g-basis | deg s-reg hom. | max deg in g-basis hom | security evaluation |
|---|---|---|---|---|---|
| 3 | 3 | 0 | 3 | 3 | $2^7$.88 |
| 5 | 3 | 2 | 3 | 3 | $2^1$3.91 |
| 7 | 3 | 0 | 4 | 3 | $2^1$9.35 |
| 9 | 4 | 0 | 4 | 4 | $2^2$4.07 |
| 11 | 4 | 0 | 4 | 4 | $2^2$8.07 |
| 13 | 4 | 1 | 4 | 5 | $2^3$1.49 |
| 15 | 4 | 1 | 5 | 6 | $2^3$4.45 |
| 17 | 5 | 0 | 5 | 6 | $2^3$7.05 |
| 19 | 5 | 0 | 5 | 7 | $2^3$9.37 |
| 21 | 5 | 0 | 5 | | $2^4$1.46 |

**Fig. 2.** Results for polynomial: $x^3 + x^2 + x * z + x$



Below is both the table and the plot of the degree of regularity for the polynomial $x^4 + x^3 + xz$, for $n < 21$. This could not be attempted any higher due to problems with sage and computing high exponents. This was made by Jack Rowland.

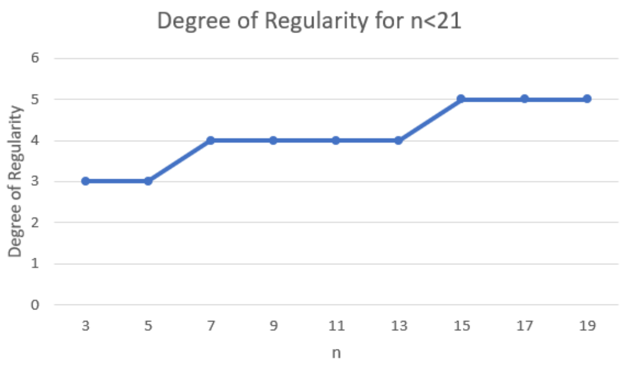
He also finds regression curve for these values. To avoid over-fitting only computed it to the third degree, so the curve is $y = 0.001x^3 - 0.0056x^2 + 0.2142x + 2.3548$. A higher degree curve results in over-fitting and a less general curve.

**Table 2.** Results for polynomial: $x^4 + x^3 + xz$

| n | deg of semi reg | deg of semi reg (hom) | max degree in gro basis | max degree in gro basis (hom) |
|---|---|---|---|---|
| 3 | 3 | 3 | 2 | 3 |
| 5 | 3 | 3 | 2 | 3 |
| 7 | 4 | 4 | 2 | 3 |
| 9 | 4 | 4 | 2 | 4 |
| 11 | 4 | 4 | 2 | 5 |
| 13 | 4 | 4 | 2 | 5 |
| 15 | 5 | 5 | 2 | 6 |
| 17 | 5 | 5 | 2 | 6 |
| 19 | 5 | 5 | 2 | 7 |

**Table 3.** Table of Security Evaluation for $E_1(x) = x^4 + x^3 + xz$

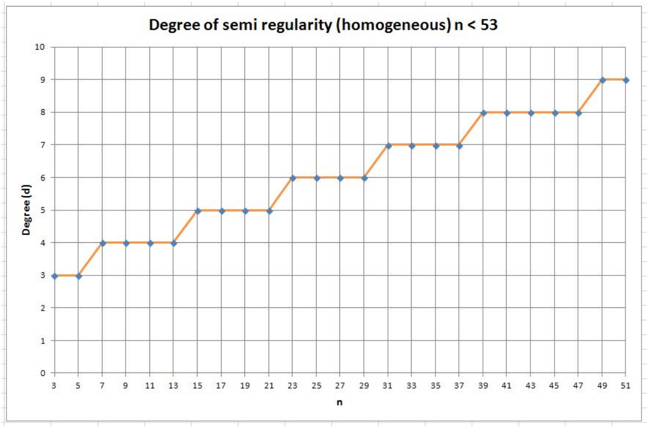
**Fig. 3.** Results for polynomial: $x^4 + x^3 + xz$



Mohammed Mirza also managed to patch the code for the security evaluation to work with values of $n > 20$. However the computation of the degree of semi-regularity became very slow for $n > 41$. Similarly, the computation of the maximum degree in the Grbner basis became very slow for $n > 21$.

Below is the plot of the degree of regularity (homogeneous) for the polynomial $xz+x^5+x^4+x^2$, for $n < 53$. The regression line has the equation $d = 0.1223n + 2.7777$.

**Fig. 4.** Results for polynomial: $xz + x^5 + x^4 + x^2$



## 7.2 General Pat

It would be interesting to tsest General Pat family of polynomials too. This task was for Marios Georgiou, Alex Nikas, Michael Suinn, unforcenetly, they have not done it.

## 7.3 MAC

Code for computing degree of regularity by Simon Bohm, Quentin Delmas and Bryce Boyd is in the appendix. They have not done complete security evaluation with estimating security level in bits. More tests needs to be done in this area.

## 7.4 Security level in bits

Maxine produced a code for estimating the security level in bits. This needs to be tested further.

```
def sec_eval_bin(n, d):
    w = log(7,2)
    T=binomial(n,d)
    for i in range(d-1,0,-1):
        T = T + binomial(n,i)
    sec_power = round(log(T^w,2),5)
    print "2^"+ str(sec_power)
for j in range(3,23,2):
    sec_eval_bin(j,5)
```

## 8 Conclusion

## References

1. Magali Bardet, Jean-Charles Faugre, Bruno Salvy: *On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations,* In ICPSS 2004.
2. Nicolas Courtois: *The security of Hidden Field Equations (HFE)*; Cryptographers' Track RSA Conference 2001, LNCS 2020, Springer, pp. 266-281. `https://pdfs.semanticscholar.org/15ef/c37e3619074803fd48376f78d15f8368dd0f.pdf`
3. Nicolas Courtois: *La sécurité des primitives cryptographiques basées sur les problèmes algébriques multivariables MQ, IP, MinRank, et HFE*, PhD thesis, Paris 6 University, 2001, in French. Available at http://www.minrank.org/phd.pdf.
4. Jean-Charles Faugère: Report on a successful attack of HFE Challenge 1 with Gröbner bases algorithm F5/2, announcement that appeared in `sci.crypt` newsgroup on the internet on April 19th 2002.
5. Neal Koblitz: "Algebraic Aspects of Cryptography"; Springer, ACM3, 1998, Chapter 4: "Hidden Monomial Cryptosystems", pp. 80-102.
6. Ming-Deh A. Huang, Michiel Kosters, Sze Ling Yeo: *Last Fall Degree, HFE, and Weil Descent Attacks on ECDLP,* In Crypto 2015, LNCS 9215, pp. 581-600, 2015.
7. Tsutomu Matsumoto, Hideki Imai: "Public Quadratic Polynomial-tuples for efficient signature-verification and message-encryption", Eurocrypt'88, Springer 1998, pp. 419-453.
8. Jacques Patarin: "Cryptanalysis of the Matsumoto and Imai Public Key Scheme of Eurocrypt'88"; Crypto'95, Springer, LNCS 963, pp. 248-261, 1995.
9. Jacques Patarin: "Hidden Fields Equations (HFE) and Isomorphisms of Polynomials (IP): two new families of Asymmetric Algorithms"; Eurocrypt'96, Springer, pp. 33-48. The extended version can be found at `http://www.minrank.org/hfe.ps`
10. Jacques Patarin: *La Cryptographie Multivariable*; Mémoire d'habilitation à diriger des recherches de l'Université Paris 7, 1999.
11. Jacques Patarin, Louis Goubin, Nicolas Courtois: *Quartz,* **1**28-*bit long digital signatures*; Cryptographers' Track Rsa Conference 2001, LNCS 2020, pp.282-297, Springer.
    **Note:** The Quartz signature scheme has been updated since, see [12].
12. Jacques Patarin, Louis Goubin, Nicolas Courtois: *Quartz,* **1**28-*bit long digital signatures*; An updated version of Quartz specification available at `http://www.cryptosystem.net/quartz/`
13. Gilles Macario-Rat, Jacques Patarin: *Two-Face: New Public Key Multivariate Schemes,* `eprint.iacr.org/2017/1210.pdf`
14. Adi Shamir, Aviad Kipnis: "Cryptanalysis of the HFE Public Key Cryptosystem"; Crypto'99. Can be found at http://www.minrank.org/hfesubreg.ps
15. Christopher J.Shallue. "Permutation Polynomials of Finite Fields". In: Monash University (2012), p. 10.

## A Code - generating public key - Simple Pat

```
import re
# Let K be extension on degree n of the field Fq where q is power of 2
def e1_enc(E1):
    q=2                                                           # <-------input
    n=15                                                         # <-------input

    K=GF(q)

    P=PolynomialRing(K, 'x', n)
    P.inject_variables()

    K2=GF(q^n)
    fx=K2.modulus()


    l=var('x%i' % n)
    MVP=PolynomialRing(K, 'x', n+1)
    MVP.inject_variables()
    xn=MVP(l)

    with localvars(fx.parent(), [l]):
        fx=MVP(fx)

    # vector of input variables (message msg)
    MSm = MatrixSpace(P,n,1)
    msg=[]
    for i in (0..n-1):
        msg.append(var('x%i' % i))
    msg=MSm(Matrix(n,1,msg))

    # basis B
    Bas=[]
    for i in (0..n-1):
        Bas.append(xn^i)



    # 0 < h < q^n, h=q^th+1, gcd(h,q^n-1)
    th=(n+1)/2                                               # <-------input
    h=(q^th)+1
    if (gcd(h,q^n-1)!=1):
        print "error"
    hp=inverse_mod(h,q^n-1)

    # affine transformation
    MSnn_2 = MatrixSpace(K,n,n)
    S=MSnn_2.random_element()
    T=MSnn_2.random_element()
```

```
# T needs to be invertible
while (T.is_invertible()==false):
    T=MSnn_2.random_element()

MSn1_2 = MatrixSpace(K,n,1)
vs=MSn1_2.random_element()
vt=MSn1_2.random_element()

# vector u computation
u=(S*msg)+vs

# vector v computation
#   v=h^h, h=q^th+1 --> v=u^q^th+u
#   in the basis:
tra=0
trb=0

for i in (0..n-1):
    tra=tra+u[i]*Bas[i]
    trb=trb+u[i]*Bas[i]^(q^th)
tr=tra*trb

exp_array = [int(s.strip('^')) for s in re.findall(r'\b\^\d+\b', str(E1))]
c = map(lambda x:1,re.findall('x(?!\^|\*)', str(E1)))
exp_array+=c
for i in (0..n-1):
    for j in (0,..len(exp_array)-1):
        tr= tr+MVP(u[i]*Bas[i]^exp_array[j])

#   computation of v, back from the basis

v=[]
for i in (1..n):
    v.append([])
tmp=expand(tr%fx)
for i in (1..n-1):
    v[i] = P(expand(tr%fx).coefficient(xn^i))
    tmp=tmp-v[i]*xn^i
v[0] = P(tmp)
v=Matrix(n,1,v)

# output y - public key
y= T.inverse()*(v-vt)
print 'y = '
print y


# ---------Security evaluation----------
# Ideal computation
L = []
```

```
    for f in y:
        print f
        print 'f[0]='
        print f[0].homogenize()
        L.append(P(f[0]).homogenize())
    #    append field polynomials in form x^2=x
    for i in (0..n-1):
        L.append(P(var('x%i' % i)^2-var('x%i' % i)).homogenize())
    print "L"
    print L
    I = Ideal(L)

    # degree of (semi)regularity and max degree in groebner_basis computation
    print I.degree_of_semi_regularity()
    print max(f.degree() for f in I.groebner_basis())
var('z')
e1_enc(x*z+x^5+x^2+x+1)
```

## B Code - generating public key - General Pat

## C Code - generating public key - MAC

```
# generates E1 for general PAT
def B_xz(q, d, x, z, alpha, beta, gamma):
    power = [] # Here are the exponents for the first two sums. In the second sum we just excluc
    if alpha[0] == 1:
        E_1 = 1 # Here E_1 = B(x,z)
    else:
        E_1 = 0
    j = 0
    k = 0
    i = 0
    # Calculate first two sums
    while q^j <= d:
        k = 0
        while q^j + q^k <= d:
            if q^j + q^k not in power:
                power.append(q^j + q^k)
                E_1 += alpha[i + 1] * x^(q^j + q^k) + beta[i] * z^(q^j + q^k)
                i += 1
            k += 1
        j += 1
    j = 0
    while q^j <= d:
        if q^j not in power:
            power.append(q^j)
            E_1 += alpha[i + 1] * x^(q^j) + beta[i] * z^(q^j)
            i += 1
        j += 1
    k = 0
    l = 0
    i = 0
    #Calculate third sum
    while q^k <= d:
        l = 0
        while q^l <= d:
            if q^k + q^l <= d:
                E_1 += gamma[i] * x^(q^k) * z^(q^l)
                i += 1
            l += 1
        k += 1
    return E_1

def has_unique_root(pol):
    if pol == 0:
        return False
    return len(pol.roots()) == 1

def is_permutation_poly(E1, q, p):
    if not has_unique_root(E1):
        return False

    for t in range(1, q - 1):
        if t % p == 0:
            continue
        red = (E1^t) % (x^q - x)
        if red.degree() > q - 2:
            return False
    return True

def is_permut_poly_brute(E1, x, field):
    results = {}
    for e in field:
        if results.get(E1.subs({x: e}), False):
            return False
        else:
            results[E1.subs({x: e})] = True
    return True

# Let K be extension on degree n of the field Fq where q is power of 2
# calculate E2 for multivariate MAC-E1
def get_E2(E1, x, y , z, t):
    B1 = E1 - y
    B2 = E1.subs(x = z, z = x^2) - t
    A = B1.resultant(B2, z)

    a = []
    for i in A.exponents():
        a.append(i[0])
    a.sort()
    return A/(x^a[0])


# modified sec val (broken)
def e1_enc(E1, q, n):
    K=GF(q)

    P=PolynomialRing(K, 'x', n)
    P.inject_variables()
    K2=GF(q^n, 'x')
    fx=K2.modulus()


    l=var('x%i' % n)
    MVP=PolynomialRing(K, 'x', n+1)
    MVP.inject_variables()
    xn=MVP(l)


    # vector of input variables (message msg)
    MSm = MatrixSpace(P,n,1)
    msg=[]
    for i in (0..n-1):
        msg.append(var('x%i' % i))
    msg=MSm(Matrix(n,1,msg))

    # basis B
    Bas=[]

    for i in (0..n-1):
        Bas.append(xn^i)

    # affine transformation
    MSnn_2 = MatrixSpace(K,n,n)
    S=MSnn_2.random_element()
    T=MSnn_2.random_element()
    # T needs to be invertible
    while (T.is_invertible()==false):
        T=MSnn_2.random_element()

    MSn1_2 = MatrixSpace(K,n,1)
    vs=MSn1_2.random_element()
    vt=MSn1_2.random_element()

    # vector u computation
    u_1=(S*msg)+vs
    u = []
    v = []
    for i in (0..n-1):
        u.append(sum(u_1[i] * Bas[i]))
        v.append(E1.subs(u[i]))


    v=Matrix(n,1,v)
    # output y - public key
    y= T.inverse()*(v-vt)
    # print 'y = '
    # print y

    # ---------Security evaluation----------
    # Ideal computation
    L = []
    for f in y:
        #print f
        #print 'f[0]='
        #print f[0].homogenize()
        L.append(P(f[0]).homogenize())
        # append field polynomials in form x^2=x
    for i in (0..n-1):
        L.append(P(var('x%i' % i)^2-var('x%i' % i)))
    # print "L"
    # print L
    I = Ideal(L)

    # degree of (semi)regularity and max degree in groebner_basis computation
    return {"dreg": I.degree_of_semi_regularity(),
           "ming": min(f.degree() for f in I.groebner_basis()),
           "maxg": max(f.degree() for f in I.groebner_basis())}

    # degree of (semi)regularity and max degree in groebner_basis computation
    return {"dreg": I.degree_of_semi_regularity(),
           "ming": min(f.degree() for f in I.groebner_basis()),
           "maxg": max(f.degree() for f in I.groebner_basis())}


# returns all possible arrays of given lenght with elements <= max_int
class PossibleArrays:
    def __init__(self, length, max_int):
        self.length = length
        self.max_int = max_int
        self.arr = [0] * length

    def __iter__(self):
        return self

    def next(self):
        i = 0
        result = self.arr[:]
        # look for next space to modify
        while i < self.length - 1 and self.arr[i] == self.max_int:
            self.arr[i] = 0
            i += 1

        if result[self.length - 1] == self.max_int + 1:
            raise StopIteration

        self.arr[i] = self.arr[i] + 1
        return result

# returns all possible array of given length with elements <= max_int
# in a random order
import random
class RandomArrays:
    def __init__(self, length, max_int):
        self.length = length
        self.max_int = max_int

    def __iter__(self):
        return self

    def next(self):
        return [random.randint(0, self.max_int) for _ in range(self.length)]


# calculate E2 for multivariate MAC-E1
def get_E2(E1, x, y , z, t):
    B1 = E1 - y
    B2 = E1.subs(x = z, z = x^2) - t
    A = B1.resultant(B2, z)

    a = []
    for i in A.exponents():
        a.append(i[0])

    a.sort()
    return A/(x^a[0])

from itertools import product

for p, d, m in product([1,2,3,4],[3,5,7],[3,5,7,9,11]):
    # Input
    q = 2^p
    n = 2 * m - 1
    amount = 1

    try:
        field = GF(q^n)
        P.<x,y,z,t>=PolynomialRing(field)

        i = 0
        while i < amount:
            alpha, beta, gamma = RandomArrays(d*d, q^n-1).next(), RandomArrays(d*d,q^n-1).next()
            E1 = B_xz(q, d, x, z, alpha, beta, gamma)

            # make polynomial univariate by setting fixed m
            E1_uni = E1.subs({z: x^q^m}).univariate_polynomial()

            # perm_poly = is_permutation_poly(E1_uni, field.order(), field.characteristic())
            perm_poly = is_permut_poly_brute(E1_uni, x, field)
            if perm_poly:
                i += 1
                print "p: %i d: %i m: %i q:%i n:%i" % (p,d,m,q,n)
                print "E1:", E1_uni, "\nE2:",  get_E2(E1,x,y,z,t).subs({t: y^2^m}), "\n"
                # e1_enc(E1_uni, q, n), "\n"
    except:
        print "fail", "p: %i d: %i m: %i q:%i n:%i" % (p,d,m,q,n)
```

## D Code - Signature and verification

```
def Verify(message, signature, pubkey):
    q=2                                                                       # <-------input
    n=5                                                                       # <-------input

    K=GF(q)

    y0, y1, y2, y3, y4 = var('y0', 'y1', 'y2', 'y3', 'y4')
    ys = [y0, y1, y2, y3, y4]

    x0, x1, x2, x3, x4 = var('x0', 'x1', 'x2', 'x3', 'x4')
    xs = [x0, x1, x2, x3, x4]

    ##-----Verification
    # pass values of the message into the public key
    pubmes_list = []
    for i in (0..n-1):
        pubmes = pubkey[i].subs({xs[i]:message[i] for i in (0..n-1)})
        pubmes_list.append(pubmes)

    # pass values of the public key into the signature, i.e. verify the signature
    ver_list = []
    for i in (0..n-1):
        ver = signature[i].subs({ys[i]:pubmes_list[i] for i in (0..n-1)})
        ver_list.append(ver)

    print 'Verification:'
    print ver_list

    if ver_list == [0,0,0,0,0]:
        print 'Valid signature'
    else:
        print 'Invalid signature'

def Sign(message):
    q=2                                                                          # <-------input
    n=5                                                                          # <-------input

    K=GF(q)

    P=PolynomialRing(K, 'x', n)
    P.inject_variables()

    K2=GF(q^n)
    fx=K2.modulus()

    l=var('x%i' % n)
    MVP=PolynomialRing(K, 'x', n+1)
    MVP.inject_variables()
    xn=MVP(l)

    with localvars(fx.parent(), [l]):
        fx=MVP(fx)

    y0, y1, y2, y3, y4 = var('y0', 'y1', 'y2', 'y3', 'y4')
    ys = Matrix([[y0], [y1], [y2], [y3], [y4]])

    MP.<ui,vi>=PolynomialRing(K, 2)
    MP.inject_variables()

    # vector of input variables (message msg)
    MSm = MatrixSpace(P,n,1)
    msg=[]
    for i in (0..n-1):
        msg.append(var('x%i' % i))
    msg=MSm(Matrix(n,1,message))

    # basis B
    Bas=[]
    for i in (0..n-1):
        Bas.append(xn^i)

    th=(n+1)/2

    # affine transformation
    MSnn_2 = MatrixSpace(K,n,n)
    S=MSnn_2.random_element()
```

```
T=MSnn_2.random_element()

# T needs to be invertible
while (T.is_invertible()==false):
    T=MSnn_2.random_element()

MSn1_2 = MatrixSpace(K,n,1)
vs=MSn1_2.random_element()
vt=MSn1_2.random_element()

# computation of vector u
u=(S*msg)+vs

tra=0
trb=0
for i in (0..n-1):
    tra=tra+u[i]*Bas[i]
for i in (0..n-1):
    trb=trb+u[i]*Bas[i]^(q^th)
tr=tra*trb

# computation of v, back from the basis
v=[]
for i in (1..n):
    v.append([])
tmp=expand(tr%fx)
for i in (1..n-1):
    v[i] = P(expand(tr%fx).coefficient(xn^i))
    tmp=tmp-v[i]*xn^i
v[0] = P(tmp)
v=Matrix(n,1,v)

# output y - public key
y= T.inverse()*(v-vt)

# computation of vector v
v = T*ys + vt

##-----Signature
#computation of E2(u,v) from E2(x,y)=x^25 + x^23 + x^20*y + x^13 +
# x^9 + x^8*y + x^7*y^2 + x^6*y + x^5*y^4 + x^5*y^2 + x^5*y^8 +
# x^3*y4 + x^2*y^3 + y^5         (simple pat, example 1)

vl= v.list()
ul = u.list()

E2_list = []
for i in (0..n-1):
    E2 = ul[i]^25 + ul[i]^23 + ul[i]^20*vl[i] + ul[i]^13 + ul[i]^9 + ul[i]^8*vl[i] + ul[i]^
    E2_list.append(E2)
# print E2's as a vector of polynomials
E2_vector = Matrix([[E2_list[0]], [E2_list[1]], [E2_list[2]], [E2_list[3]], [E2_list[4]] ])
print ('Signature:')
print E2_vector
```

## E Code - More blocks

```
#############################################
# Polynomial Class
#############################################
import numpy as np

class MultiPoly:
    def __init__(self, c = [], v = [], e = [], copy = None):
        if copy != None:
            self.coeff = np.array(copy.coeff)
            self.var = np.array(copy.var)
            self.exp = np.array(copy.exp)
        else:
            self.setCoeff(c)
            self.setVar(v)
            self.setExp(e)

    #Iterator stuff
    def __iter__(self, index = 0):
        self.index = index
        return self

    def __getitem__(self,key):
        try:
            return(self.coeff[key], [(v, self.exp[i][key]) for i,v in enumerate(self.var)])
        except IndexError:
            return np.nan

    def next(self):
        n = self.getTerm(self.index)
        if n is np.nan:
            raise StopIteration
        else:
            self.index += 1
            return n

    #Setter functions
    def setCoeff(self, c):
        if isinstance(c, list):
            self.coeff = np.array(c)
        else:
            raise ValueError('Coefficients must be in array form')

    def setVar(self, v):
        if isinstance(v, list):
            self.var = np.array(v)
        else:
            raise ValueError('Variables must be in array form')

    def setExp(self, e):
        if isinstance(e, list):
            if len(e) > 0 and isinstance(e[0],list):
                self.exp = []
                for entry in e:
                    self.exp.append(entry)
            else:
                self.exp = np.array([np.array(e)])
        else:
            raise ValueError('Exponents must be in array form')


    #Gets the term at index in polynomial, fails if imbalanced lists
    def getTerm(self, index):
        try:
            return(self.coeff[index], [(v, self.exp[i][index]) for i,v in enumerate(self.var)])
        except IndexError:
            return np.nan

    #Evaluate the polynomial with given arguments
    def evaluate(self,**kwargs):
        value = 0
        for term in self:
            tempVal = 1
            for v in term[1]:
                try:
                    tempVal *= kwargs[v[0]]^v[1]
                except Exception as e:
                    raise type(e)('Variable: ' + str(e) + ' not provided to evaluate function')
            value += term[0]*tempVal
        return value

    #Symbolically evaluate polynomial
    def symEval(self,**kwargs):
        for arg in kwargs:
            if isinstance(kwargs[arg], tuple) and arg in self.var:
                varIndex = int(np.where(self.var == arg)[0])
                dupIndex = int(np.where(self.var == kwargs[arg][1])[0]) if kwargs[arg][1] in sel
                self.var[varIndex] = kwargs[arg][1]
                for i,term in enumerate(self):
                    if term[1][varIndex][1] != 0:
                        self.coeff[i] *= kwargs[arg][0]
                        self.exp[varIndex][i] *= kwargs[arg][2]
                if dupIndex is not None:
                    self.exp[dupIndex] = [z[0]+z[1] for z in zip(self.exp[dupIndex], self.exp[va
                    self.exp = np.delete(self.exp, varIndex, 0)
                    self.var = np.delete(self.var, varIndex)
            else:
                raise ValueError('Symbolic evaluation args must have form <varName>=(coeff,newVa


    def sortByExp(self,varIndex=0):
        for passnum in range(len(self.exp[varIndex])-1,0,-1):
            for i in range(passnum):
                if self.exp[varIndex][i]>self.exp[varIndex][i+1]:
                    for exp in self.exp:
                        temp = exp[i]
                        exp[i] = exp[i+1]
                        exp[i+1] = temp
                    temp = self.coeff[i]
                    self.coeff[i] = self.coeff[i+1]
                    self.coeff[i+1] = temp

    #Print function
    def printPoly(self):
        i = 0
        term = self.getTerm(i)
        outStr =""
        for term in self:
            if term[0] == 0:
                continue
            if term[0] != 1:
                outStr += str(term[0])
            for v in term[1]:
                if v[1] != 0:
                    outStr += str(v[0]) + "^{" + str(v[1]) + "}"

            outStr += " + " if outStr != "" else "1 + "

        print outStr[:len(outStr) - 2]




import math
import itertools
import numpy as np

def genB(d, q, numVars = 2):
    ub = floor(math.log(d, q))+1

    vals = []
    for i in range(ub^numVars):
        vals.append([floor(i/ub^mod) % ub for mod in range(numVars)])

    BExp = [[0] for _ in range(numVars)]
    if d >= 2:
        for i in range(numVars):
            for j in range(numVars):
                BExp[j].append(1 if i == j else 0)
    doneSums = []

    for val in vals:
        #if len([(i,v) for i,v in enumerate(val) if v != 0]) == 1:
        #    for i in range(numVars):
        #        BExp[i].append(q^val[i] if val[i] != 0 else 0)
        valSum = sum([q^v for v in val])
        if valSum <= d:
            for i in range(numVars):
                BExp[i].append(q^val[i])
            if valSum not in doneSums:
                doneSums.append(valSum)
                for i in range(numVars):
                    for j in range(numVars):
                        BExp[j].append(valSum if i == j else 0)

    #BCoeff = list(np.random.randint(q, size=len(BExp[0])))
    BCoeff = [1]*len(BExp[0])

    BVar = ['x']
    for i in range(numVars - 1):
        BVar.append("z"+str(i))

    return(BCoeff, BVar, BExp)


#############################################
# Resultant Attempt
#############################################
import sympy as sp
from sympy.matrices import Matrix
from sympy import *

def resultant(poly1, poly2):
    if isinstance(poly1, MultiPoly) and isinstance(poly2, MultiPoly):
            #Yucky indexing...
            dim1 = poly1[-1][1][0][1]
            dim2 = poly2[-1][1][0][1]
            totDim = dim1 + dim2
            sylArray = np.zeros((totDim, totDim))

            coeff1 = []
            for i in range(dim1 + 1):
                if i in poly1.exp[0]:
                    coeff1.append(poly1.coeff[np.where(poly1.exp[0]==i)][0])
                else:
                    coeff1.append(0)

            coeff2 = []
            for i in range(dim2 + 1):
                if i in poly2.exp[0]:
                    coeff2.append(poly2.coeff[np.where(poly2.exp[0]==i)][0])
                else:
                    coeff2.append(0)

            for i in range(dim1):
                for j in range(dim1+1):
                    sylArray[j+i][i] = coeff1[j]
                for j in range(dim2+1):
                    sylArray[j+i][i+dim1] = coeff2[j]

            np.linalg.det(sylArray)



    else:
        raise ValueError('Parameters must be MultiPoly objects')


#############################################
# Timing Code
#############################################
import matplotlib.pyplot as plt
import time
import numpy as np

alpha = beta = gamma = [1] * 1000
d = 6
q = 2
runs = 10
timeArray = []

for i in range(runs):
    b1 = B_xz(q, d, x, z, alpha, beta, gamma)
    b2 = B_xz(q, d, z, x^(q^(i+1)), alpha, beta, gamma)

    res1 = b1 - y
    res2 = b2 - t
    start = time.time()
    E2 = res1.resultant(res2, z)
    end = time.time()
    timeArray.append(end - start)
```

```
t = np.arange(0.0, runs, 1)
plt.plot(t, timeArray)
plt.xlabel('Number of Variables')
plt.ylabel('Time (s)')
plt.title('Time to Calculate E2')
plt.savefig("e2Group3Part4.pdf")
plt.show()
```