Project CS4095 about

"Hidden Field Equations" (HFE)

Supervisors:
Patrick Fitzpatrick (Department of Mathematics) &
Simon Foley (Department of Computer Science)

- Project Report -

by Christopher Wolf

Abstract

This project deals with a public key cryptography system called "Hidden Field Equations" (HFE). This system can be used both for encryption and signature. The aim of the project was first to understand and describe HFE and second to implement an HFE version in Java.

HFE uses polynomials over finite fields. It is recommended to implement it over $\mathrm{GF}(2^n)$ for $127 \leq n$ with an univariant polynomial of degree $25 \leq d \leq 33$ and d to be odd [HFE]. While public key operations (that is, encryption and signature verification) are quite fast, private key operation (that is, decryption and signature generation) are much slower. The most expensive step is solving a polynomial equation of the form $P(x) = y$ over $\mathrm{GF}(2^n)$. This project uses the Berlekamp Trace Algorithm [LN86] to solve this equation. Another drawback is the very large public key size of about 100kb for $n = 129$. This key size grows $O(n^3)$ and is therefore about 1Mb for $n = 257$. Moreover, there are attacks which threaten the security of HFE.

On the other hand, it is possible to vary HFE in a way so these attacks do no longer apply. One of these variations (called "HFE-") involves hiding some of the public equations, another variation (called "HFEv") introduces some more variables, the so-called "vinegar variables". Both versions are considered to be secure [HFE, MG01]. Moreover, HFE is neither based on the factorization problem nor on the discrete logarithm problem. Hence no algorithm which solves there two problems is a threat for HFE. Last but not least, HFE is claimed to be faster than RSA [HFE96]. Versions of HFE called "Quartz", "Flash" and "SFlash" are currently under consideration to become a European signature standard [HFE, NESSIE, QFS].

This project implemented HFE- in Java for $n = 67, 129$ and $d = 33$. It also produced a "toy" version of HFE for $n = 6$ and $d = 20$ which made the study of HFE- easier.

As this project was rather a research than an engeneering project, rapid prototyping with some elements of extreme programming [KB99] was used to obtain the code. Both versions together consists of approx. 14,000 lines of Java code. The most lines deal with public key generation, while encryption and decryption were more or less straight forward.

After an overview on the in the classes, this report describes some parts of the code in greater detail and concentrates especially on key generation for HFE.

Contents

1 Introduction

New ideas in cryptography emerged during the last millennia mainly to protect military secrets. During the last decades, the use of cryptography has shifted to more commercial applications. This particularly includes using the Internet for B2B or B2C trading models and can be summed up under the term “eCommerce”. Another use is to secure sensitive data either within a company or within a private computer by encryption methods.

The aim of this project was to investigate an new approach to public key cryptography - “Hidden Field Equations” (HFE) and to implement them in Java. This report is organized as follows: This first section will provide a brief overview of public key cryptography and the corresponding underlying problems. Section 2 gives an introduction on the mathematics related to HFE. In Section 3 describes known vulnerabilities and attacks on HFE. How these attacks can be avoided will be explored in Section 4. Then we will move to implementation issues which have been taken in consideration for this project in Section 5. A Java implementation is discussed in Section 6. This Section gives also a more detailed explanation about public key generation. The report concludes with Section 7 with some comments on the project and gives an outlook on HFE in practice.

1.1 Public Key Cryptography

A milestone practical cryptography was the paper [DH76] from Diffie and Hellman about public key cryptography. In 1976 they introduced the idea of not using one single, secret key for both encryption and decryption but one key for encryption (public key, denoted K) and one key for decryption (private key, denoted k). This opened the door to a completely new type of cryptography and helped to overcome the serious problem of distributing keys.

To explain the overall idea of public key cryptography, we will make use of the three characters Alice, Bob and Eve. In this setting Alice wants to send a message to Bob and Eve wants to eavesdrop.

It is assumed that everybody knows Bob's Public Key K_B but only Bob knows his private key k_b. When Alice wants to send a message m to Bob, she performs $m' := E(K_B, m)$ to encrypt it. Here $E(\cdot,\cdot)$ and K_B are publicly known. Moreover, the computation of $E(\cdot,\cdot)$ is feasible. Therefore Alice can perform $E(K_B, m)$. On the other hand, $D(\cdot,\cdot)$ is publicly know, too, but not k_b. So only Bob can compute $m = D(k_b, m')$ and thereby obtain the original message m.

It is necessary to postulate for all given public/private key pairs (K_X, k_x) and all messages m that encryption can be reversed, that is $m = D(k_x, E(K_X, m))$. Additionally, we assume that it is not feasible to compute k_x knowing K_X, or m knowing m' in this scheme.

The system described above can be used both for encryption and signature. Interestingly, there are currently more schemes that can be used for signature than there are for encryption [HFE96]. One example of a signature *only* system is the graph isomorphism problem combined with the Hamilton cycle problem [HAC96].

In the following sections we will have a look at some systems which can be used both for encryption and decryption.

1.2 Factorization

One of the first such systems was RSA. It exploits, firstly, the fact that multiplying two integers is easy but factoring their product is difficult and, secondly, the Fermat-Euler theorem, that is, $a^{\phi(n)} \equiv a \pmod{n}\ \forall a, n \in \mathbb{Z}$ [HAC96].
These two facts are used in the following key generation algorithm:

1. Generate $p, q \in \mathbb{Z}$ prime
2. Compute $n := pq$ and $\phi = (p-1)(q-1)$
3. Select $e \in \mathbb{Z}$: $\gcd(\phi, e) = 1$
4. Compute $d \in \mathbb{Z}$ with $de \equiv 1 \pmod{\phi}$

In this system, the key pair is:

- Public Key $K := (e, n)$
- Private Key $k := (d, n)$

The encryption can be done by computing $m' := E(m, K) := m^e \pmod{n}$ only using publicly known information. The decryption makes use of $k = (d, n)$ and thereby computes $D(k, m') := (m')^d \equiv m^{ed} \equiv m \pmod{n}$.

If it is not feasible to factor n, the function $\phi(n)$ cannot be computed and thereby it is not possible to compute e knowing $K = (d, n)$. RSA is therefore said to rely on the factorization problem.

The contrary, namely that breaking RSA also means that the factorization problem is solved, is not true. There might be ways to break RSA without solving the factorization problem.[HFE96]

The Blum-Goldwasser system is also based on the factorization problem but makes use of randomness [HAC96, p. 308-310].

Another public key systems that relies on the factorization problem is the Rabin public key encryption scheme. It is proven to be equivalent to the factorization problem [HAC96, p.292].

1.3 Discrete Logarithm

ElGamal is another important system that is widely used for public key cryptography. It relys on the problem of solving discrete logarithms, that is, solving $a^x \equiv b \pmod{n}$ for given $a, b, n \in \mathbb{Z}$.
Key generation works as follows:

1. Generate prime $p \in \mathbb{Z}$
2. Find a generator $\alpha \in \mathbb{Z}$ for $\mathbb{Z}_p^*$
3. Select a random integer $a \in \{1, \ldots, p-2\}$
4. Compute $\alpha^a \mod p$

Here, the key pair is:

- Public Key $K := (\alpha, \alpha^a, p)$
- Private Key $k := (a, p)$

To perform encryption, compute

$$m' := E(m, K) := (\gamma, \delta) := (\alpha^k \mod p, m(\alpha^a)^k)$$

where $k \in_R \{1, \ldots, p-2\}$ denotes a randomly selected integer. The message can be revealed using the function

$$D(k, m') := (\gamma^{-1}) \cdot \delta \equiv \alpha^{-ak} m \alpha^{ak} \equiv m \pmod{p}$$

$$\text{as } \gamma^{p-1-a} \equiv \gamma^{-a} \equiv \alpha^{-ak} \pmod{p}.$$

Decryption requires the knowledge of a. As long as it is not practical to solve $\alpha^a \equiv \alpha \pmod{p}$, ElGamal is secure. Thus, ElGamal relys on the discrete logarithm problem. On the other hand, to break ElGamal it may not be necessary to solve the discrete logarithm problem.

Elliptic curve cryptography makes also use of the discrete logarithm problem but uses a different group, namely a group generated by an elliptic curve [BSS99].

1.4 Miscellaneous Public Key Schemes

The McEliece public key encryption system [HAC96, pp. 298-299] is based on error-correcting codes and has resisted cryptanalysis up to now. Both the public and the private key operations are relatively fast, but the public key is very large (approx. 1Mb for the recommended security parameters). Therefore, it is not used in practice [HAC96].

Another problem, not used in practice, is based on the knapsack problem, that is, to fit objects of different value in a container and obtain a maximal value in the container. The objects have discrete size and the container can only hold a discrete number of objects. The use of this problem was first proposed by Merkle and Hellman [HAC96, pp. 300-306]. Their scheme was broken [HAC96]. The system of Chor and Rivest, however, which is also based on the knapsack problem, is not broken yet [HAC96]. With the recommended security parameters, the public key is approx. 10 kb and therefore considered to be too big for practical use [HAC96].

The list of problems given in this introduction is far from being complete. Its purpose is to give an idea of problems that are suitable for public key encryption.

1.5 HFE Problem: MinRank

Another problem, which can be used for public key cryptography is called "MinRank" problem. In this problem, a set of randomly generated, quadratic equations over $\mathbb{F}_2$ is given and a solution for this set of equations is required. The problem can be generalized for any finite field ($|\mathbb{F}| \geq 2$) and any degree $d \geq 2$. MinRank is NP complete [HFE96, GJ79, p. 251].

Having these publicly known equations, HFE uses a trapdoor consisting of two affine transformations S and T over $\mathbb{F}$ and one private polynomial $P(x)$ over $\mathbb{E}$ of degree $d := \partial P(x)$. So the private key is the triple $k := (S, P, T)$. The public key consists mainly of n polynomials in n variables, each over $\mathbb{F}$, that is, $K := (p_1, \ldots, p_n)$.

The next sections will give a more detailed description of different aspects of HFE, the underlying mathematics and how to implement HFE.

2 HFE: Mathematical Background

In "Hidden Field Equations" (HFE), a finite field with $q := |\mathbb{F}|$ elements, characteristic p (with p prime) and an extension field $\mathbb{E}$ over $\mathbb{F}$ are used. The extension field is generated by the irreducible polynomial $i(t)$ over $\mathbb{F}$. This polynomial $i(t)$ has degree $n := \partial i(r)$. Extension field $\mathbb{E}$ is also identified with vector space $\mathbb{F}[t]/i(t)$ and some operations of HFE are not done in $\mathbb{E}$ but in $\mathbb{F}[t]/i(t)$. This means that every $e \in \mathbb{E}$ can be seen as a vector $(e_1, \ldots, e_n) : e_i \in \mathbb{F}$ and, moreover, as a polynomial of degree $n-1$ at most in $\mathbb{F}[t]/i(t)$, where addition is normal polynomial addition and multiplication is done modulo $i(r)$.

The private polynomial P is over the extension field $\mathbb{E}$. It has degree $d := \partial P$ and some restrictions on the powers that are allowed for P. In brief: P can only have powers with Hamming weight 2 at most (for the case $q = 2$). In symbols:

$$
\begin{aligned}
P &: \mathbb{E} \to \mathbb{E} \\
P(x) &= \sum c^{(i)} x^{h_i}, \text{where} \\
& \quad c^{(i)} \in \mathbb{E}, h_i \leq d, \\
& \quad h_i \neq h_j \ \forall i \neq j, \\
& \quad h_i = \begin{cases} 0, & & \text{(this is the constant term)} \\ q^a, & a \in \mathbb{N}_0 & \text{(these are the linear factors)} \\ q^b + q^c, & b, c \in \mathbb{N}_0 & \text{(these are the quadratic factors)} \end{cases}
\end{aligned}
$$

The purpose of this restriction on powers h_i is to keep public polynomials $(p_1, \ldots, p_n)$ small. To make the private key operations simpler, it is possible to restrict leading coefficient (that is, $c^{(i)}$ with corresponding $h_i = d$) to be 1, that is, to deal only with a monic polynomial for P.

The two affine transformations used in HFE are over $\mathbb{F}$. They are denoted S and T and can be represented by one $n \times n$ matrix and one n-dimensional vector each.

As we see in Figure 1, the public key $K = (p_1, \ldots, p_n)$ is used to bypass the private key $k = (S, P, T)$. Despite MinRank being NP complete, HFE is not [HFE96]. This situation compares with the relationship between RSA and the factorization problem or ElGamal and the discrete logarithm problem.

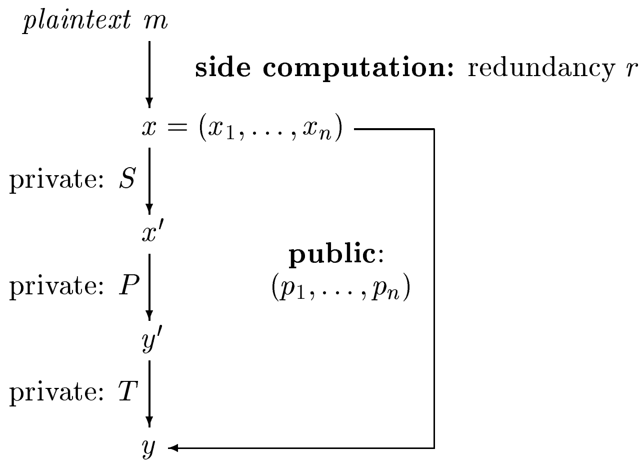

Figure 1: Overall Structure of HFE for Encryption with Ciphertext (y, r)

2.1 Encryption and Decryption of Messages using Private Key

In this section we will investigate message encryption and decryption using private key $k = (S, P, T)$. Please see Figure 1 for the overall structure of these operations.

Since the affine transformations S and T are over $\mathbb{F}$, it is necessary to transfer the message m from $\mathbb{E}$ to $\mathbb{F}^n$ before encrypting it. This is done by treating m as the vector $(x_1, \ldots, x_n) \in \mathbb{F}^n$. Thus, we no longer think about the extension field as a field but as an n-dimensional vectorspace over $\mathbb{F}$.

To encrypt $(x_1, \ldots, x_n)$, we first apply S, giving the result x'. At this point, note that x' is transformed from $\mathbb{F}^n$ to $\mathbb{E}$ in order to apply the private polynomial $P \in \mathbb{E}[x]$. The result $y' := P(x)$ is element of $\mathbb{E}$. Once again, y' is transformed to vector $(y'_1, \ldots, y'_n)$, transformation T is applied giving $y = (y_1, \ldots, y_n)$. So the final output is (y, r). The added redundancy r will be discussed later.

To decrypt y, the above steps are done in reverse order. This is possible as the private key (S, P, T) is known. The crucial step in deciphering is not inversion of S and T, rather computation of x' in equation $P(x') = y'$. As polynomial P has degree d, there are up to d different solutions for this equation [LN86, HFE96]. Addition of redundancy to the message provides an error-correcting effect and makes it possible to select the right m from the set of solutions x'. This redundancy is added at the first step (see Figure 1) and must be done by sender and recipient using the same function.

Example

The following example shows how to encrypt and decrypt a message in $\mathbb{F} = \mathrm{GF}(2)$ and $\mathbb{E} = \mathrm{GF}(4)$ generated by $i(\gamma) = \gamma^2 + \gamma + 1$.

Moreover, for this example, let

$$s' := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, s := \begin{pmatrix} 1 \\ 1 \end{pmatrix}, t' := \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, t := \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Set $S := (s', s)$, $T := (t', t)$ and $P(x) := x^2 + 1$ to obtain the private key $k := (S, P, T)$.

Encryption We encrypt message $m = (1,0)^T$ over $\mathbb{F}^2$.

1. Apply S:

$$\begin{aligned} s'm + s &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{aligned}$$

2. Transfer result from $\mathbb{F}^2$ to $\mathbb{E}$: $(0,1)^T \to 1$.
3. Apply P: $1^2 + 1 = 1 + 1 = 0$
4. Transfer result from $\mathbb{E}$ to $\mathbb{F}^2$: $0 \to (0,0)^T$.
5. Apply T:

$$\begin{aligned} t'(0,0)^T + t &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{aligned}$$

So, $(1,0)^T \to (0,1)^T$ in this example.

Decryption For decryption, we have to reverse each step:

1. Apply T^{-1}:

$$\begin{aligned} t'^{-1}[(0,1)^T + t] &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \left[\begin{pmatrix} 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right] \\ &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{aligned}$$

2. Transfer result from $\mathbb{F}^2$ to $\mathbb{E}$: $(0,0)^T \rightarrow 0$.

3. Solve $P(x) = 0$: By inspection of all possible values $0, 1, \gamma, \gamma^2$ we see that only $x = 1$ satisfies this equation:

   - $P(0) = 0^2 + 1 = 1$
   - $P(1) = 1^2 + 1 = 0$
   - $P(\gamma) = \gamma^2 + 1 \equiv \gamma$
   - $P(\gamma^2) = \gamma^4 + 1 \equiv \gamma + 1$

4. Transfer result from $\mathbb{E}$ to $\mathbb{F}^2$: $1 \rightarrow (0,1)^T$.

5. Apply S^{-1}:

$$\begin{aligned} s'^{-1}[(0,1)^T + s] &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \left[\begin{pmatrix} 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right] \\ &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{aligned}$$

So we obtain the original message $m = (1,0)^T$ after decryption with the private key $k = (S, P, T)$.

2.2 Private Key: Solving $f(x) = y$

As outlined above, deciphering $m' = (y, r)$ requires solving of polynomial equations $f(x') = y'$ over the finite field $\mathbb{E}$. This problem has been well studied and [HFE96, pp. 8-10] mentions six different algorithms. We will use a special case of the Berlekamp trace algorithm. The reason for using Berlekamp in this project is that it is efficient if q is small (see [HFE96, p. 9] and [AM93, p.23/24]). As this implementation deals with $q = 2$, we can be sure that this condition is true. For our configuration, we have expected running time $O(d^2n^3 + d^3n^2)$ where d is the degree of P and n is the degree of the irreducible polynomial $i(t)$, that is the dimension of $\mathbb{E}$ over $\mathbb{F}$. The worst case complexity is $O(d^2n^4 + d^3n^3)$ [HFE96, pp. 8/9]. The description of solving polynomial equations in this section is based on techniques from [LN86, AM93, HFE96, DK97].

To apply the Berlekamp trace algorithm and rule out trivial cases, our equation $f(x) = y$ must fulfil the following properties:

1. We deal with $f(x) - y = 0$ instead of $f(x) = y$,
2. $f(x)$ is monic, that is, leading coefficient is 1,
3. $f(x)$ is non-trivial, that is, degree $\partial f(x) > 1$, and
4. $f(x) - y$ has no repeated factors.

As the first three properties can be achieved very easily, we may assume for the rest of this section - without loss of generality - that $f(x)$ fulfils these requirements. The fourth demand, however, needs further consideration.

To deal with (4) we recall that for our purpose, namely root finding, we are not interested in all irreducible factors of $f(x) - y$ but only in monic linear factors, that is $x - e$ with $e \in \mathbb{E}$. Therefore we exploit the fact, that the product of these factors can be computed quite easily in a finite field using the following equation[LN86]

$$\prod_{e\in\mathbb{E}} x - e = x^q - x \text{ where } q := |\mathbb{E}|$$

Hence, we compute $g(x) := \gcd(x^q - x, f(x) - y)$ and apply the Berlekamp algorithm to $g(x)$ instead of $f(x)$. As $g(x)$ is the gcd of $f(x)$ and $x^q - x$, it will contain all monic, linear factors of f but nothing more than these.

Descriptions of the Berlekamp Trace Algorithm can be found in [AM93, DK97, LN86]. They cover the more general case of factoring a polynomial instead of finding the roots of a given polynomial. But as $g(x)$ has only monic factors, applying this algorithm to $g(x)$ will yield only the roots of $f(x) - y$.

2.3 Public Key: Generation and Encryption

Recall that the public polynomials $p_1, \ldots, p_n$ must be computed in such a way that they can be used to apply the private key (S, P, T) to message m without revealing this private key.

To compute these public polynomials $(p_1, \ldots, p_n)$, we transfer P from $\mathbb{E}$ to $\mathbb{F}^n$. This means we think about $x \in \mathbb{E}$ as $(x_1, \ldots, x_n) \in \mathbb{F}^n$ and

$c^{(i)} \in \mathbb{E}$ as $(c_1^{(i)}, \ldots, c_n^{(i)}) \in \mathbb{F}^n$. Given this perspective, we no longer have one polynomial in one variable, but n polynomials in n independent variables, namely $p_1(x_1, \ldots, x_n), \ldots, p_n(x_1, \ldots, x_n)$ as we have input $(x_1, \ldots, x_n) \in \mathbb{F}^n$ and output $(y_1, \ldots, y_n) \in \mathbb{F}^n$ for P.

To compute these n polynomials, we first have to transform $(x_1, \ldots, x_n)$ and $(c^{(i)}, \ldots, c^{(i)})$ using S. This transformation is not necessary from a strictly mathematical point of view as S is an affine transformation and therefore all operations in $\mathbb{E}$ can also be done in $\mathbb{E}' \xrightarrow{S} \mathbb{E}$ and vice versa (the two field $\mathbb{E}'$ and $\mathbb{E}$ are isomorphic). ¿From a more practical point of view, we need to transform both the coefficients and also $(x_1, \ldots, x_n)$ as we want to compute public polynomials that perform S, T and P. From a cryptographer's point of view, both S and T prevent P and m being revealed.

Having applied S to the vectors, we now apply P to these vectors. Exploiting the Euler-Fermat Theorem, we see that x_i^q is a linear transformation in $\mathbb{F}$ and therefore $x_i^q \longrightarrow x_i$ for $x_i \in \mathbb{F}$. Moreover, we make use of the binomial theorem for finite fields

$$(a+b)^q = (a+b)^{p^n} = a^{p^n} + b^{p^n} \ \forall a, b \in \mathbb{E}$$

where p denotes the characteristic of the extension field $\mathbb{E}$ and $q = p^n = |\mathbb{E}|$.

Additionally we know that

$$h_i = \begin{cases} 0, & \\ q^a, & a \in \mathbb{N}_0 \\ q^b + q^c, & b, c \in \mathbb{N}_0 \end{cases}$$

and therefore know that polynomials $(p_1, \ldots, p_n)$ are of degree 2 at most. As the last step we have to apply T to these polynomials to compute our public key.

To encrypt a message m using this public key, we transfer it to $\mathbb{F}^n$ and apply $(p_1, \ldots, p_n)$.

Additionally we have to compute the redundancy r to make unique decryption possible. As outlined in Section 2.1, we need this redundancy r to select the correct solution x of $f(x) = y$ and hence obtain the correct message m.

2.4 Message Signature

In addition to encryption / decryption, HFE can also be used for signing a message m. As for decryption, we use the fact that it is computationally not feasible to get a solution $(x_1, \ldots, x_n)$ for

$$\begin{aligned} y_1 &= p_1(x_1, \ldots, x_n) \\ y_2 &= p_2(x_1, \ldots, x_n) \\ &\ldots \\ y_n &= p_n(x_1, \ldots, x_n) \end{aligned}$$

when $(p_1, \ldots, p_n)$ are quadratic in $x_1, \ldots, x_n$ without the trapdoor (S, P, T) (see Section 1.5).

Considerations

By using the private key $k = (S, P, T)$, this problem reduces finding a solution to equation $P(x) = y$ where P has degree d. As we have seen in Section 2.2, this is feasible. Unfortunately, in general $P(x)$ is not a surjection and therefore $\exists y : P(x) \neq y \ \forall x \in \mathbb{E}$. Keeping this in mind, we cannot find a solution $(x_1, \ldots, x_n)$ for each MinRank problem. This means, if we do not succeed in finding a solution x for a certain y in $P(x) = y$, we have to try with another y until we succeed. In HFE, the number of y we have to try is small [HFE96].

Computation

To explain signature generation and verification, we will restrict to the case that the message m was reduced to 128 bits using an arbitrary one-way-hash-function $h(x)$, e.g. SHA-1 or MD5. Furthermore, we restrict to $q = 2$ and $n = 163$. This means that 128 bits of y are fixed (namely the hash of message m) and 35 bits in y "free". We can use any value for these 35 bits and especially vary them if $P(x) = y$ has no solution.

After computing $h(m)$, we try to find a solution for $P(x) = y$. If we do not succeed, we try with another value for the 35 "free" bits. If we succeed, we publish x to be the signature for message m.

Verification

When somebody wants to verify that message m was signed with HFE, this person uses the public key, that is, $K = (p_1, \ldots, p_n)$, and computes the 128 bit hash of the message m, that is, $h(m)$. After this precomputation, this person compares:

$$\begin{aligned} h_1 &= p_1(x_1, \ldots, x_{163}) \\ h_2 &= p_2(x_1, \ldots, x_{163}) \\ &\vdots \\ h_{128} &= p_{128}(x_1, \ldots, x_{163}) \end{aligned}$$

If all 128 equations are satisfied, the signature is valid. Otherwise, it is not. Note that only 128 of the 163 public equations are necessary to verify the signature. Therefore in a signature scheme, only a limited number of equations will be published (see also Section 4).

3 Some Attacks on HFE

In this section we will examine two possible attacks on HFE. This section concludes that basic HFE scheme must be regarded as broken.

3.1 Linear Attack

For the linear attack described in this section [HFE96], we assume that we know y, y' and d where y =HFE(x) and y' =HFE($x + d$). Here HFE is one invocation of the HFE system as described above and we know the difference d between the original messages $x, x + d$ but not x itself.

By calculating the difference between y and y' we can compute x as outlined below. First we subtract the two equations:

$$\begin{aligned} y_i - y_i' \quad = \quad & (a_{i,0} - a_{i,0}) + \\ & + a_{i,1}(x_1 - x_1 - d_1) + \cdots + a_{i,n}(x_n - x_n - d_n) + \\ & + a_{i,1,1}(x_1^2 - x_1^2 + 2x_1 d_1 - d_1^2) + \\ & + a_{i,1,2}(x_1 x_2 - x_1 x_2 - x_1 d_2 - x_2 d_1 - d_1 d_2) + \cdots + \\ & + a_{i,n,n}(x_n^2 - x_n^2 + 2x_n d_n - d_n^2) \end{aligned}$$

In this equations, all the nonlinear factors for x_i vanish and we obtain a system of linear equations:

$$\begin{aligned} y_i - y_i' = \quad & a_{i,1}d_1 + \cdots + a_{i,n}d_n + a_{i,1,1}(2x_1 d_1 - d_1^2) + \\ & + a_{i,1,2}(-x_1 d_2 - x_2 d_1 - d_1 d_2) + \cdots + \\ & + a_{i,n,n}(2x_n d_n - d_n^2) \end{aligned}$$

The resulting n equations are linear in the unknowns $x_1, \ldots, x_n$ and can therefore be solved in polynomial time.

This attack can be avoided by padding the vector x with random bits or by introducing a publicly known, linear attack resistant function $x = f(m)$, e.g. DES, AES with publicly known key k. This function $f(x)$ can then be applied to x before encrypting it [HFE96].

3.2 Quadratic Attack

In contrast to Section 3.1, this attack [HFE96] makes assumptions about the private polynomial P, especially about the fact that it exists. This means that this attack does not try to solve the underlying, NP complete MinRank problem but makes the assumption that there is a trapdoor. Moreover, this attack makes use of public polynomials $(p_1, \ldots, p_n)$ and assumes that the cipher text $y = (y_1, \ldots, y_n)$ is known.

To apply this attack, public polynomials $K = (p_1, \ldots, p_n)$ are rephrased by introducing new variables $X_{j,k} := x_j x_k, j \leq k$. Counting the number of quadratic terms in the public key K gives $\frac{n(n-1)}{2}$ as an upper bound for the number of new variables $X_{j,k}$. We assume for the rest of this section that this is also a lower bound and therefore the cardinality $\#X_{j,k} = \frac{n(n-1)}{2}$. To justify the lower bound, we compute the probability for $i = 1, \ldots, n$ equations to have $a_{i,j,k} = 0$ as leading coefficient for *all* $x_j x_k$ (with j, k fixed). This probability is $(1/q)^n$ for $q = |\mathbb{F}|$. For this calculation we assume random distribution for the coefficients $a_{i,j,k}$. With $q = 2$ and $n = 129$ (both reasonable values for q, n, see Section 5.1), we obtain the negligible value $(1/2)^{129} \approx 10^{-40}$ that a certain $X_{j,k}$ cancels. Our lower bound is therefore reasonable.

So by replacing quadratic terms $x_j x_k$ with variables $X_{j,k}$, we get n linear equations in $y_1, \ldots, y_n$ knowns and $x_1, \ldots, x_n, X_{1,1}, X_{1,2}, \ldots, X_{n,n}$ unknowns, that is, $n + \frac{n(n-1)}{2} = \frac{n(n+1)}{2}$ unknowns and n knowns. We assume that these n equations are pairwise independent. This means that we "restrict" our search space from q^n to $q^{[n(n-1)]/2}$.

To compensate for this "blow up", we need $\frac{n(n-1)}{2}$ more equations. Furthermore, all of our equations must be pairwise independent. Combining $y_j = p_j(x_1, \ldots, x_n)$ and $y_k = p_k(x_1, \ldots, x_n)$ in a linear way, that is, $ay_j + by_k = ap_j + bp_k$ for $a, b \in \mathbb{F}$ is not an option as these equations depend on $(y_1 = p_1, \ldots, y_n = p_n)$ and therefore do not restrict our search space. But we can combine $y_j = p_j$ and $y_k = p_k$ in a *non linear* way, e.g. by computing $y_j \cdot y_k = p_j \cdot p_k$. As quadratic terms in $X_{j,k}$ prevent us from solving these equations, we need to concentrate on the case that all quadratic terms in $X_{j,k}$ cancel and we therefore obtain a linear equation in $\frac{n(n+1)}{2}$ unknowns.

For a randomly generated system of quadratic equations, the probability of fulfilling both conditions is very poor. But having the special trapdoor P, the chances are much better. The key point is that it is possible to express P in different, pairwise independent ways over $\mathbb{F}^n$. As long as these ways of expressing P are invariant under affine transformations, this structure of P will "leak" into the public polynomials $(p_1, \ldots, p_n)$ and can therefore be exploited in an attack. It is important to notice that only certain polynomials P have this property and that only certain ways of expressing P are invariant under affine transformations. [HFE96] gives a list of equations that are invariant under affine transformations, e.g. S and T. The list is exhaustive for equations of degree 2 and 3.

We also have to notice, that we are not only concerned about getting exactly $\frac{n(n+1)}{2}$ equations this way but that any $\lambda \approx \frac{n(n+1)}{2}$ equations will do as this restricts our search space to

$$q^{\frac{n(n+1)}{2}-\lambda} \text{ and for } \lambda \approx \frac{n(n+1)}{2} \text{ we notice } q^{\frac{n(n+1)}{2}-\lambda} < q^n$$

So this attack on HFE is more efficient than exhaustive search in our whole message space $\mathbb{F}^n$ but needs special polynomials P. These polynomials are considered to be "weak keys".

To justify the last $<$ sign in the equation above, we have to see that we can combine at most $\frac{n(n-1)}{2}$ polynomials p_j and p_k in a quadratic way and therefore $\lambda \leq \frac{n(n-1)}{2}$.

A generalization of this attack would be combining three or more equations. This is not recommended in [HFE96] as the number of computations needed to gain these equations is higher than the number of computations for exhaustive search of m.

3.3 General Attacks on HFE

Changing our point of view, we will look on possible attacks at HFE from a more general perspective. Using [MG01, HFE01, QFS] we see that there are four different types of attacks possible against HFE:

1. Exhaustive search for m.

2. Revealing the private key using the public key.

3. Solving the quadratic equations
   $y_1 = p_1(x_1, \ldots, x_n)$
   $\ldots$
   $y_n = p_n(x_1, \ldots, x_n)$
   as if there were no be a trapdoor.

4. As above but using the fact that there is a trapdoor.

Attacks (1), (3) are not practical as the running time is normally exponential. Moreover, the existence of a polynomial time algorithm for (3) would solve the MinRank problem and therefore prove P=NP, which is unlikely. Therefore we concentrate on (2) and (4). The best known attacks to date have complexity $O(n^{O(\log d)})$ [HFE01]. They are polynomial in time and space if d is fixed. This is usually the case in HFE and therefore basic HFE is considered to be broken [HFE01, MG01].

4 HFE Variations

The attacks (2) and (4) from Section 3 need special conditions. These conditions are certainly fulfiled in basic HFE. But it is possible to vary HFE in a way that the attacks described above no longer apply. According to [MG01, HFE01], these variations on HFE are secure.

4.1 HFE-

The first variation, denoted HFE-, reduces the number of polynomials that are published, that is, $k : 1 \leq k \ll n$ of the n public polynomials $(p_1, \ldots, p_n)$ are kept secret.

As outlined in Section 2.4, keeping polynomials secret is not a problem for a signature algorithm. For decryption, some bits of message m are missing in HFE-. Therefore it is necessary to check all possible messages $m_1, \ldots, m_{q^k}$. The number of possible messages is q^k as k equations are removed and there are q possibilities for each of this equations. Using redundancy r it is possible to select the correct message m_i.

HFE- is therefore a good choice for a signature scheme but slows down decryption. Anyway, if k, q are small enough (e.g. $k = 8$ and $q = 2$), it is feasible for decryption, too.

4.2 HFEv

The second variation "HFEv" introduces "vinegar" variables $x'_1, \ldots, x'_v$ over $\mathbb{F}$. This changes the underlying field for S. The affine transformation S here is $S : \mathbb{F}^{n+v} \longrightarrow \mathbb{F}^{n+v}$. The coefficients c_i also depend on these new variables, therefore the definition of P changes to:

$$\begin{aligned} P &: \mathbb{E} \to \mathbb{E} \\ P(x, x'_1, \ldots, x'_v) &= \sum c_i(x'_1, \ldots, x'_v) x^{h_i}, \text{where} \\ & \quad c_i(x'_1, \ldots, x'_v) \in \mathbb{E}, h_i \leq d, \\ & \quad c_i(x'_1, \ldots, x'_v) \text{ is linear for} \\ & \quad h_i \text{ linear and quadratic for} \\ & \quad h_i \text{ quadratic.} \end{aligned}$$

The values of these vinegar variables are chosen at random in a signature scheme, the public polynomials are now $p_i(x_1, \ldots, x_n, x'_1, \ldots, x'_v)$.

For an encryption scheme, HFEv is not possible as the values of $x'_1, \ldots, x'_v$ are not known before $y = P(x)$ is solved. Therefore it is not feasible to introduce vinegar variables in an encryption scheme.

As for HFE-, the attacks on basic HFE do not apply to HFEv.

4.3 HFEv-

For signature schemes, it is possible to combine both HFE- and HFEv, that is, to reduce the number of public equations and to introduce vinegar variables. This system is denoted HFEv-.

4.4 Further Variations

In [HFE96], some more variations are suggested. We will give a short outline of some of these modifications.

It is possible to introduce some random polynomials $q_1, \ldots, q_k$ and mix these polynomials with $p_1, \ldots, p_n$ in transformation T. In a signature scheme, these polynomials will reduce the probability of finding $x_1, \ldots, x_n$ that satisfy $h(m) = HFE(x)$ by $(1/q)^k$. If k is small, this is feasible. In an encryption scheme, these polynomials are no problem as it is possible to ignore them while solving $P(x) = y$. However, equation $k+n \ll \frac{n(n+1)}{q}$ must be satisfied. Otherwise it is possible to introduce new variables $X_{i,j} := x_i x_j \ \forall i,j : i \leq j$ (see Section 3.2), solve this linear system of equations and obtain the message m after resubstituting $X_{i,j} = x_i x_j$.

Secondly, it is possible to have a private polynomial P in k independent variables. If this polynomial P is not chosen at random, but in a way that it is possible to solve equation $(y_1, \ldots, y_k) = P(x_1)$, then $(y_1, \ldots, y_k) = P(x_1, x_2)$, $\ldots$, $(y_1, \ldots, y_k) = P(x_1, \ldots, x_{k-1})$ and finally $(y_1, \ldots, y_k) = P(x_1, \ldots, x_k)$, it is feasible to solve all of these equations. As the underlying extension fields are smaller, the private key operations are faster. Unfortunately, the overall extension field $\mathbb{E}$ no longer has prime dimension and therefore subfield attacks might be possible.

Thirdly, it is possible to replace the irreducible polynomial $i(t)$ with a reducible polynomial $r(t)$. The extension $\mathbb{F}[t]/r(t)$ is therefore no longer a field but a ring. This means that operations in $\mathbb{F}[t]/r(t)$ might be faster for particular choices of $r(t)$ (e.g. many coefficients $r_i = 0$). Unfortunately, the proof for the upper bound of the number of solutions for $P(x) = y$ in a field does no longer apply and we might get $s \gg d$ solutions for this equation. Although [HFE96, p. 30] claims that the linearized polynomial algorithm can still be used to solve $P(x) = y$ over $\mathbb{F}[t]/r(t)$, we assume that using $r(t)$ instead of $i(t)$ does not improve HFE.

The variations to introduce more then one branch (see [HFE96, p. 30]), to restrict the values for S and T on a particular subfield (see p.30), and public polynomials with degree 3 instead of 2 are not discussed in this report as they are of no use in practice according to [HFE96].

4.5 HFE in Practice

This section will conclude with a look at how HFE is used in practice, namely the versions currently under consideration as a European signature standard. The algorithms submitted to [NESSIE] are called Quartz, Flash and SFlash [QFS]. This section starts with Quartz:

Quartz

In terms of HFE, it is a HFEv- scheme (see Section 4). The parameters are

n	d	q	k	v
103	129	2	3	4

In this notation, n is the dimension of the extension field $\mathbb{E}$, parameter d denotes the degree of the private polynomial P while q is the number of elements of the underlying field $\mathbb{F}$. As Quartz is an HFEv- scheme, k denotes the number of public equations removed and v is the number of "vinegar variables".

In contrast to "standard" HFEv-, there are 4 invocations of the private key $k = (S, P, T)$. They are combined in a so called "Feistel-Patarin" scheme (FPS). The purpose of FPS is to bring down the signature length from 400 bits to 128 bits. A detailed description of FPS can be found in [MG01, QFS].

The key sizes in Quartz are [QFS]:

- Public Key: 71 kb
- Private Key: 3 kb

Quartz is considered to be secure (see [QFS] and [MG01]).

Flash, SFlash

In contrast to Quartz, Flash and SFlash are not really based on HFE but more closely related to the original Matsumoto-Imai scheme [MI88]. In contrast to this system, there are many public equations removed. As a result, Flash and SFlash are secure although the Matsumoto-Imai scheme is considered to be broken.

All three use a polynomial of the form

$$f(A) = A^{1+q^\theta} \text{ with } \theta = 11 \text{ for Flash and SFlash}$$

For SFlash and Flash, f is a bijection and f^{-1} can be computed easily (e.g. in a smart card). Additionally, f is publicly known so the private key only consists of the two affine transformations (S, T).

The other parameters for both algorithms are given in the following table:

Parameter	SFlash	Flash
q	$128 = 2^7$	$256 = 2^8$
n	37	
k	11	
θ	11	
$f(A)$	$A^{128^{11}+1}$	$A^{256^{11}+1}$
Signature	259 bits	296 bits
Private Key Size	0.35 kb	2.75 kb
Public Key Size	2.2 kb	18kb

The very small key sizes in SFlash are due to a "trick", namely to restrict the coefficients of transformations S, T and polynomials $(p_1, \ldots, p_{26})$ to a subfield $\mathbb{F}_2$ of $\mathbb{F}_{2^7}$. This idea is explained in [HFE96, QFS].

As for Quartz, there are currently no successful attacks known against Flash and SFlash. Anyway, as [MG01] points out, the parameters in Flash and SFlash are carefully chosen so the currently known attacks do not apply to them. In contrast, Quartz has four different lines of defence: the large degree d, the introduction of vinegar variables, removing of some public equations, and the Feistel-Patarin scheme. Therefore Flash and SFlash are certainly less secure than Quartz.

5 Configuration

As we have seen in Sections 3 and 4, it is important to implement a secure version of HFE to avoid attacks. This aim was achieved in this project.

HFE is patented by Bull [QFS] and to the knowledge of the author, its source code is not public available. This implementation is therefore based on publicly available documentation, e.g. [HFE96, HFE, QFS].

To assure portability, the implementation is done in Java. Having the implementation being available in Java also allows easy integration into the Java Cryptographic Framework [JCE14].

The scheme implemented is a HFE- scheme as it should be used both for encryption and signature. HFEv cannot be used for encryption (see Section 4.2). The remainder of this section will discuss different parameters for HFE- and their impact on key size.

5.1 Security Parameters

As we have seen in Section 2, there are different parameters in HFE that change the level of security but also the speed of the HFE operations. As a matter of fact, there are the following parameters in HFE:

Field Size q of Field $\mathbb{F}$

This should be small for two reasons. First of all, it speeds up the finite field operations, e.g. the Berlekamp trace algorithm. Secondly, to keep d small, as d is a function in $q^m + q^n$ for $m, n \in \mathbb{N}$. Increasing q results in an exponential growth of d. On the other hand, a smaller q will result in many coefficients for the public polynomials and therefore a bigger public key K. In this implementation, $q = 2$ as recommended in [HFE96].

Dimension n of the Extension Field $\mathbb{E}$

According to [HFE], n should be prime to avoid subfield attacks although there are no attacks currently known that make use of the fact that $\mathbb{E}$ has subfields [HFE]. Increasing n also increases the number of public polynomials and therefore the public key size. In terms of speed, increasing n is better than increasing d as the public key operations are much faster and a bigger public key size does not slow them down too much. [HFE] recommends to have $n \geq 127$. In this implementation, $n = 67, 129$. The value $n = 67$ is chosen to show how HFE speeds up using different values of n.

Degree d of Polynomial P

Having a small degree d for the private polynomial P results in fast private key operations, in particular solving the equation $P(x) = y$. Simulations in [HFE96] suggest the form $d = 2k+1$ for $k \in \mathbb{N}$ as there is a security "quantum leap" between $d = 2k$ and $d = 2k + 1$ but the private key operations have roughly the same speed. In [HFE], a value for d between 25 and 33 is recommended. In this implementation, $d = 33$.

Structure of Polynomial P

As outlined in Section 3.2, some polynomials are considered to be weak. As it is difficult to check if a certain polynomial is weak or not, I followed recommendations from [HFE96, p. 12] and chose the following private polynomial for my HFE implementation:

$$\begin{aligned}P(x) = x^{33} &+ c_{32}x^{32} + c_{24}x^{24} + c_{20}x^{20} + c_{18}x^{18} + c_{17}x^{17} + c_{16}x^{16} + c_{12}x^{12} \\ &+c_{10}x^{10} + c_9x^9 + c_8x^8 + c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x\end{aligned}$$

Here $c_i \in \mathbb{E}$ and all powers of x have Hamming weight ≤ 2.

5.2 Key Size

The security parameters discussed above lead to the key sizes given in Figure 2. As you can see in this figure, the size of private key is negligible compared with public key.

n	Public	Private
67	19kb	1.4kb
129	134kb	4.6kb

Figure 2: Key Size in HFE with $q = 2$ and $d = 33$

Generally speaking, the public key size depends on the number of coefficients for the public polynomials and also on the number of elements in the finite field $q = |\mathbb{F}|$. We obtain as size of the public key:

$$q \cdot n((n+1)(n+2))/2 = O(q \cdot n^3)$$

The private key size depends on the number of coefficients in the private polynomial P, the affine transformations S and T, and the number of elements in the finite field $q = |\mathbb{F}|$. We compute an upper bound that also depends on the degree d of P:

$$q[n(d+1) + 2 \cdot (n^2 + n)] = O(q \cdot (dn + n^2)) = O(q \cdot n(d+n))$$

Having these parameters chosen and the key sizes in mind, we will start looking how HFE- can be expressed in terms of Java classes.

5.3 Class Overview HFE4_3

The prototype for this project was produced using a condensed version of "extreme programming" (see Section 6.1.1). An overview of all classes for this prototype can be seen in Figure 3. This picture is not a class hierarchy but shows how the different classes use each other. The class `Check`, for example, is used in all other classes but uses no other class and is therefore on the very bottom of the picture. However, the class `Multi4Power` makes use both of `Poly4MultiLinear` and `Poly4Max` and is therefore on the top of both of them.

As the code of this project was rather evolved than planed, some of the class definitions are not "perfect" in a sense that they would not have occurred in a strictly planning approach (e.g. water fall model). One example is the class `Poly64Max` as this class does not only hold operations for the finite field $\mathbb{E} = \mathrm{GF}(64)$ but also operations for polynomials over $\mathrm{GF}(64)$ with a fixed number of coefficients.

On the following pages we will see a brief overview of all classes that are used in this toy version:

Check As Java 1.3 does not provide an automatic assertion checker, I implemented one in this class. The assertion checker for HFE is quite similar to the assertion facility of Java 1.4.

Field4 (Section 6.2.1) This class encapsulates the operations for a finite field with 4 elements. The current field is generated by the polynomial $f(t) = t^2 + t + 1$.

Field64_Tab (Section 6.2.1) This class encapsulates the look-up tables for multiplication, division, and inverses. The current field is generated by the polynomial $f(t) = t^3 + t^2 + 2t + 3$.

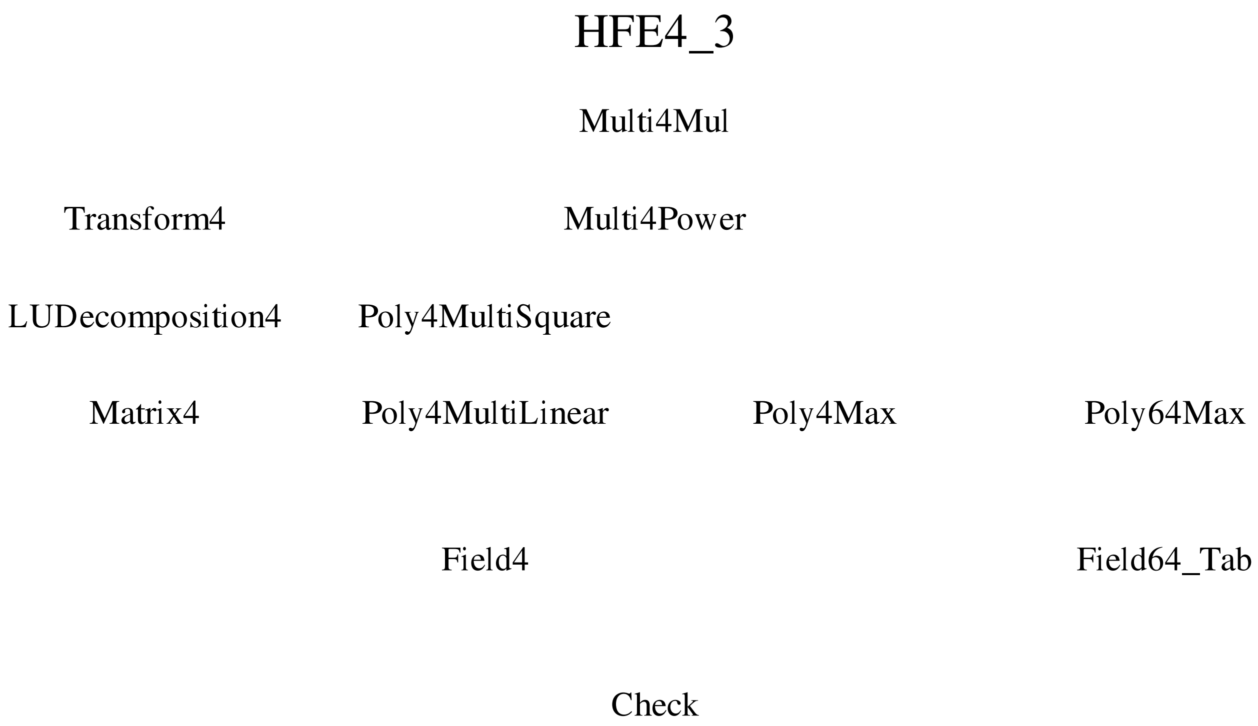

Figure 3: Classes for $\mathbb{E} = \mathrm{GF}(4^3)$

HFE4_3 This class is the heart of the "toy version" of HFE and implements key generation, encryption, decryption, message signature and signature verification.

LUDecomposition4 (Section 6.2.4) Because inverting the two affine transformations S and T involves solving a matrix equation $A \cdot x = b$ with $A \in \mathbb{F}^{n \times n}$ and $x, b \in \mathbb{F}^n$, this class provides the facility of using the LU decomposition algorithm. It is based upon the [JAMA] package.

Matrix4 As `LUDecomposition4`, this class is based on the [JAMA] package. It implements Matrix operations over GF(4) and is used to implement S and T.

Multi4Mul (Section 6.2.3) This class deals with multi-variable polynomials and a general modulus provied to the constructor. Basically, this class calculates multiples of two polynomials or one polynomial and one element of GF(64). All calculations are done in F_4. Every entry is a polynomial of the form $f_i(x_1, x_2, \ldots, x_n) = g^i(b_0 + b_1x_1 + b_2x_2 + \cdots + b_nx_n)$ where $x_i, b_i, g_i, x_i \in$ GF(4). The reduction is done in the variable g.

Poly4Max (Section 6.2.2) The polynomials in this class are in one single variable, namely x and of degree `maxLen` at most.

Poly4MultiLinear (Section 6.2.3) In contrast to `Poly4Max`, the polynomials in this class have n different variables but the degree of each term is bounded by 1. Both coefficients and unknowns come from GF(4).

Poly4MultiSquare (Section 6.2.3) As polynomials in `Poly4MultiLinear` are of degree 1 at most, the members of this class are quadratic. Both coefficients and unknowns come from GF(4).

Poly64Max (Section 6.2.2) This class has two purposes: First it implements finite field operations in GF(64). This field is generated over GF(4) by the polynomial $f(x) = x^3 + x^2 + 2x + 3$. Second, it implements polynomials in one variable with fixed upper degree and is therefore similar to `Poly4Max`.

Transform4 The purpose of this class is to encapsulate the two affine transformations (that is, S, T in terms of HFE). This means that this class allows to computes $y = M * x + m$ with matrix $M \in \mathrm{GF}(4)^{n \times n}, \det(M) \neq 0$ and vector $m \in \mathbb{F}^n$. The vectors $x, y \in \mathbb{F}^n$ can serve either as output or input. In addition, this class is used during public key generation (see 6.3).

5.4 Class Overview HFE2_n

As for Figure 3 in Section 5.3, the classes in Figure 4 should be read from the bottom to the top, that is, each class which uses another class can be found above this class. For example, `Poly2_nMax` makes use of `Check` and `Field2_n` but not of `Matrix2`.

This description will not give a complete class overview as for HFE4_3 but only point out how the classes outlined above had to be changed to adapt HFE to $\mathbb{F} = \mathrm{GF}(2)$ and $\mathbb{E} = \mathrm{GF}(2^n)$ for $n = 67, 129$.

Field2_n (Section 6.2.1) The first change is the introduction of a class `Field2_n` that hosts the finite field operations for different values n.

RootFinding (Section 6.2.5) In addition to this change, a new class `RootFinding` was introduced. It implements methods to solve polynomial equations $P(x) = y$ over $\mathbb{E}$ and follows the algorithm outlined in Section 2.2. It also makes use of the interface `CheckRoot`.

In the following sections we will have a closer look on the operations used in `HFE4_3` and `HFE2_n`. The examples are usually taken from `HFE4_3` but are also true for `HFE2_n`.

HFE2_n

Multi2Mul

RootFinding

Transform2

Multi2Power

CheckRoot
(Interface)

LUDecomposition2

Poly2MultiSquare

Matrix2

Poly2MultiLinear

Poly2Max

Poly2_nMax

Field2

Field2_n

Check

Figure 4: Classes for $\mathbb{E} = \mathrm{GF}(2^n)$

6 Implementation

This section will give an overview on the ideas and methods used during this project in Section 6.1. In Section 6.2 it will give detailed descriptions of selected items of the project. Section 6.3 will give further explanation of the key generation process. In Section 6.4 the performance results will be presented and discussed.

6.1 Engineering

6.1.1 Extreme Programming

As this project was rather a research than an engineering project, it was hard to plan it. Therefore, extreme programming [KB99] was adapted for this project. Moreover, the first version was a "toy" version `HFE4_3` that works on $\mathbb{F} = \mathrm{GF}(4)$, $\mathbb{E} = \mathrm{GF}(4^3) = \mathrm{GF}(64)$. This was used to produce the final version for $\mathbb{F} = \mathrm{GF}(2)$ and $\mathbb{E} = \mathrm{GF}(2^n)$ for $n = 67, 129$.

According to [KB99, pp. 53-61], extreme programming consists of the following parts:

1. *Planing game:* balance business priorities and technical estimations.
2. *Small releases:* short production cycle rather than complex changes.
3. *Metaphor:* use a simple shared story to guide development.
4. *Simple design:* remove unnecessary complexity.
5. *Testing:* each module must pass all test before it is used.
6. *Refactoring:* rewrite code after it works.
7. *Pair Programming:* all code is written by two people on one machine.
8. *Collective ownership:* code can be changed by any person in the team.
9. *40-hour week:* do not work more than 40 hours a week as a rule.
10. *On-site customer:* during coding, include a real user on the team.
11. *Coding standards:* all code follows the same rules.

As this was a one man project, many of these points did not apply (obviously, pair programming). However, small releases, simple design, testing, refactoring, and coding standards proved to be valueable during this project.

Small Releases In terms of this project, small releases was read as working step by step from bottom to top. Every module was fully tested before it was used. So the finite field operations were finished before working on the polynomials classes and so on.

Simple Design This was vital, too, as it prevented the project from having nice but useless features. It concentrated on implementing HFE rather than working on new algorithms or special Java features.

Testing Looking at the whole project, testing was certainly the most vital part during this project. Nearly half of the lines in each class are test cases.

Refactoring As Java is rather slow, refactoring during this project mainly concentrated on making things faster. The simple design helped to achieve this goal.

Coding Standards It might look ridiculous to have coding standards in a single programmer project but this project consists of approx. 14,000 lines of code. So having similar names for similar things in different classes helped the coding process.

6.1.2 Assertion Checking

Despite extreme programming does not stress assertion checking so much, it proved to be important during this project.

Assertions are assumptions the programmer makes for his code, e.g. after calling method `sort(testArray)`, the programmer assumes that `testArray` is sorted (example taken from [JAAS]). Java 1.4 introduces the possibility of checking assumptions during the programme run [J14N].

As the final release of Java 1.4 was not available when this project started but I felt that having an assertion checking facility would help me to find flaws in my code quicker, I decided to write my on assertion checker class:

```
class Check {
   static void assert(boolean statement) {
     if (!statement) throw
       new IllegalArgumentException("Assertion wrong");
   }

   static void assert(boolean statement, String text) {
     if (!statement) throw
```

```
          new IllegalArgumentException("Assertion wrong: " + text);
7.    }
8. }
```

As you see from the code above, the class is only 8 lines long. The two `static` functions `assert(boolean)` and `assert(boolean, String)` constain an `if` statement each (see lines 3 and 6). This `if` throws an exception when parameter `statement` is not true. As both methods throw an `IllegalArgumentException`, it is not necessary to declare this exception within Java.

The second method (lines 5-7) provides the same functionality as the first but has additional parameter `text`. This parameter can be used to print information about the reason for the exception.

For the sake of speed there are two Perl scripts that automatically comment and uncomment any statement beginning with the text `Check.assert(`. This makes it possible to switch the assertion check on and off on the level of source code. To my knowledges, this is not possible with a compiler directive as Java lacks directives similar to, for example, `#ifdef` in C and C++.

Class `Check` gave me important hints while debugging my code. Note that assertion checking was switched off during performance measurements.

6.1.3 Javadoc

A side effect of choosing Java as implementation platform is that I could make use of `javadoc`, which is part of Java 1.3. This gave me the opportunity to produce documentation for this project while writing the code [JDOC].

As output format for our documentation I chose HTML as I felt that this gave me high flexibility (e.g. by including HTML tags into our documentation) and small file size for the documentation at the same time.

6.2 Detailed Descriptions

While Sections 5.3 and 5.4 gave a brief overview on the overall structure of this project, this section will go into details of different aspects on a deeper level. It roughly follows the columns in Figures 3 and 4.

6.2.1 Finite Field Operations

The whole project uses four different fields: GF(2), GF(4), GF(4^3), and GF(2^n). The elements of the first three fields are stored in one `integer` each. For $\mathrm{GF}(2^n)$, an array of `integer` is used. For all these fields, addition was done using `xor`. As all these fields have characteristic 2, subtraction and addition are the same. Hence, subtraction is done using `xor`, too. However, multiplication and division require further work.

For $\mathrm{GF}(4)$ and $\mathrm{GF}(4^3)$, the results of multiplication, division and the multiplicative inverse are stored in an arbitrary table which gives a nice speed up.

For $\mathrm{GF}(2^n)$, this is no longer feasible as the table become too large for $n = 67, 129$: in the case of $n = 67$ and multiplication alone this table has $2^{2 \cdot 67} \approx 10^{40}$ entries, 67 bits each. Strategies such as Zech logarithms (see [KH90]) were taken into consideration. For $n = 67$ we get $2^{67} \approx 10^{20}$ entries, which is feasible. But for $n = 129$ we obtain $2^{129} \approx 10^{39}$ that is out of range again.

Therefore I had to implement finite field operations using standard algorithms from [LD00]. The only "non standard" algorithm is division in $\mathrm{GF}(2^n)$ that uses the recent algorithm described in [ShC01]. The field elements from $\mathrm{GF}(2^n)$ are stored in coefficient representation.

To make the overall structure flexible, $\mathrm{GF}(2^n)$ is triggered by the two constants `genPoly` and `extension`. They hold the generating polynomial and the dimension of the extension field. To have have one central point to change from $\mathrm{GF}(2^{67})$ to $\mathrm{GF}(2^{129})$ (and vice versa), there is a central "switch" for the different values of all constants in `HFE2_n`.

6.2.2 Polynomials in One Variable

This section explains how polynomials over the extension field $\mathbb{E}$ are stored in this implementation.

For polynomials over this extension field $\mathbb{E}$ we chose a coefficient representation, that is, for every power i of x we store the corresponding coefficient $a_i \in \mathbb{E}$ in an array. We implemented all the operations that are used in HFE. These are: addition/subtraction of polynomials, multiplication, multiplication modulo another polynomial $M(x) \in \mathbb{E}[x]$ and exponentiation modulo $M(x)$.

Polynomial multiplication is done using a standard $O(n^2)$ algorithm [CLR96]; a later version might use [CLR96, p. 798] that provides $O(n \ln^2 n)$ by using a Fast Fourier Transformation like algorithm.

Calculation of the polynomial $P(x)$ modulo $M(x)$ is done using a "paper and pencil" like approach, as the following pice of code shows:

```
static Poly64Max mod_additive = new Poly64Max();

void mod (final Poly64Max b) {
  Check.assert(!b.isZero());
  while (b.degree() <= this.degree()) {
    mod_additive.copyShift(b,this.degree()-b.degree());
    mod_additive.mulFactor(divNum(
         value[degree()], b.value[b.degree()]));
    add(mod_additive);
  }
}
```

In the first line of mod, assertion checker from class Check ensures that polynomial b is not zero. For the sake of speed, this assertion is not checked during performance measurements. The while loop is repeated as long as b has less or equal degree than this, that is, as long as it is possible to reduce this using b. In the first version of this method, the first line in the while loop generated a new object in every invocation. For the sake of speed, the static variable mod_additive was introduced. The second line in the while loop ensures that the leading coefficient in both this and additive are the same and therefore the following line will reduce the degree of this by at least one. It might look strange that addition of additive and this has this effect but we have to keep in mind that we work in GF(2^n). In this class of fields, every element is its own additive inverse and therefore addition and subtraction are the same.

6.2.3 Multivariable Polynomials

The representation of the multivariate polynomials exploits the fact that they are of degree 2 at most. Figure 5 illustrates how the coefficients are stored in the coefficient array.

Array Index	0	1	2	3	4	5	6	7	8	9
Variable	1	x_1	x_2	x_3	x_1^2	x_1x_2	x_1x_3	x_2^2	x_2x_3	x_3^2
Coefficient	a_0	a_1	a_2	a_3	a_{11}	a_{12}	a_{13}	a_{22}	a_{23}	a_{33}

Figure 5: Coefficient Array of Multivariate Polynomial, $n = 3$

In general, the number of terms stored in the coefficient array is $\tau(n) := ((n+1)(n+2))/2$ elements. Therefore $O(n^2)$ space is required. The polynomial operations are based on this structure and thus $\tau(n)$ is also the lower bound in terms of time.

For performance reasons, multivariate polynomials were implemented in two separate classes for each version of HFE. One class supports polynomials of degree 1, with operations performed in $O(n)$ space and time. Another class (using the coefficient array from Figure 5) supports polynomials of degree 2, with operations performed in $O(n^2)$. This distinction provides a marked speedup during key generation, as a significant component of the key generation process is done using multivariate polynomials of degree 1.

To see how the ideas discussed in this section are reflected in Java, we will inspect the multiplication of two linear polynomials over GF(4) that gives a quadratic polynomial over GF(4):

```
/** Calculates this = a*b
      in terms of polynomial multiplication. a and b must have the
      same number of variables. The current value of this is discarded.
  */
public void mul(Poly4MultiLinear a, Poly4MultiLinear b) {
  Check.assert(a.giveVarNum() == b.giveVarNum());
  Check.assert(a.giveVarNum() == varNum);

  int [] A, B;
  A = a.coeffs();
  B = b.coeffs();

  // assume that we are working on a (varNum+1)*(varNum+1) matrix
  // and reduce that to a triangle matrix for the terms which are
  // not on the diagonal
  int k = 0;    // k <= j all the time
  int j = 0;
  for (int i = 0; i < coeff.length; i++) {
   // do we have a quadratic term?
    if (k == j)
      coeff[i] = Field4.mul(A[k],B[k]);
    else
      coeff[i] = Field4.add(Field4.mul(A[k],B[j]),Field4.mul(A[j],B[k]));
    j++;
    if (j > varNum) {
      k++;
      j = k;
    }
  }
}
```

This algorithm is based on the observation that multiplying two polynomials $a(x_1, \ldots, x_n) = a_0 + a_1x_1 + \cdots + a_nx_n$ and $b(x_1, \ldots, x_n) = b_0 + b_1x_1 + \cdots + b_nx_n$ can be expressed in terms of a matrix:

$\times$	b_0	b_1x_1	$\ldots$	$b_{n-1}x_{n-1}$	b_nx_n
a_0	a_0b_0	$a_0b_1x_1$	$\ldots$	$a_0b_{n-1}x_{n-1}$	$a_0b_nx_n$
a_1x_1	$a_1b_0x_1$	$a_1b_1x_1^2$		$a_1b_{n-1}x_1x_{n-1}$	$a_1b_nx_1x_n$
$\vdots$	$\vdots$		$\ddots$		$\vdots$
$a_{n-1}x_{n-1}$	$a_{n-1}b_0x_{n-1}$	$a_{n-1}b_1x_1x_{n-1}$		$a_{n-1}b_{n-1}x_{n-1}^2$	$a_{n-1}b_nx_{n-1}x_n$
a_nx_n	$a_nb_0x_n$	$a_nb_1x_1x_n$	$\ldots$	$a_nb_{n-1}x_{n-1}x_n$	$a_nb_nx_n^2$

We find that the terms on the diagonal all have the form x_i^2 including the case 1, that is, no variable. Moreover we see that the terms on the diagonal occur only once.

The situation is different for terms that are not on the diagonal. They always appear twice but with two different coefficients, for example, $a_1b_0x_1$ and $a_0b_1x_1$ or $a_{n-1}b_1x_1x_{n-1}$ and $a_1b_{n-1}x_1x_{n-1}$.

Exploiting the fact that we store our terms for result c in order

Array Position	0	1	$\ldots$	n	$(n+1)$	$(n+2)$	$\ldots$	$(\tau(n)-1)$
Coefficient	c_0	c_1	$\ldots$	c_n	$c_{1,1}$	$c_{1,2}$	$\ldots$	$c_{n,n}$

we get the very compact algorithm outlined above.

Lines 2 and 3 are for debugging purposes only and ensure that the two linear polynomials `a` and `b` have the same number of variables (that is, `giveVarNum()`) as our quadratic polynomial (that is, `varNum` in line 3).

The steps in lines 4-6 are for speed purposes only as accessing array elements directly is faster than accessing these elements using an arbitrary class method.

Multiplication is done in the for loop, that is, lines 10-20. The key point is line 11 where we distinguish between elements that are on the diagonal, that is, depend on the result of *one* multiplication $a_k \cdot b_k$ and elements that are not on the diagonal and therefore depend on the sum of *two* multiplications $a_k \cdot b_j + a_j \cdot b_k$.

While counter `i` is used to go through the coefficient array of an object of type `Poly4MultiSquare`, that is, a quadratic polynomial, counters `j` and `k` go through the coefficients of polynomials `a` and `b`. As it is enough to go through the upper triangle of the matrix outlined above to obtain our result, we make sure $k \leq j$ in our for loop. Our incrementation of `j` and `k` in lines 15-19 preserves this equation $k \leq j$ and also makes sure $j, k \leq n$ as both polynomials `a` and `b` have exactly $n+1$ entries in their coefficient arrays `A` and `B`.

For the sake of speed there is no new object `Poly4MultiSquare` generated but the coefficient array `this.coeff` is overwritten with the result of the multiplication.

6.2.4 LU Decomposition

LU decomposition for a matrix $A \in \mathbb{F}^{n \times n}$ gives matrices $L, U \in \mathbb{F}^{n \times n}$ with $A = L \cdot U$. Here L is a lower triangular matrix and U an upper triangular matrix. This means $i < j \implies l_{i,j} = 0$ for coefficients $l_{i,j}$ from matrix L and $i > j \implies u_{i,j} = 0$ in matrix U. This LU decomposition always exists for a given matrix $A \in \mathbb{F}^{n \times n}$ [RH96]. As Jama restricts the diagonal in L to $l_{i,i} = 1$, this LU decomposition is moreover unique for a given A [JAMA].

After an LU decomposition is computed, it can be used to solve equations like

$$A \cdot x = b \text{ with } A \in \mathbb{F}^{n \times n} \text{ and } x, b \in \mathbb{F}^n$$

In HFE this problem occurs to invert affine transformations S and T.

Moreover, as $\{M \in \mathbb{F}^{n \times n} \mid \det M \neq 0\}$ forms a group under matrix multiplication [DW98], and $\det AB = \det A \cdot \det B$ [RH96],

$$\det L, \det U \neq 0 \implies \det LU \neq 0 \implies \det A \neq 0$$

This problem occurs in HFE to gain if matrix M in transformations S, T is singular or non-singular. If M is non-singular, transformations S and T are affine.

So LU decomposition can be exploited to solve equation $Ax = b$ without computing A^{-1} and also to find out if $\det A \neq 0$. This approach is reasonable as computing A^{-1} and $\det A$ is computationally more expensive than computing the LU decomposition of A [RH96].

After the LU decomposition of A is computed, equation $Ax = b$ can be solved exploiting the fact that solving $Ly = b$ is computational easy. As matrix L is triangular, y_1 (that is, this is the first row of vector y) depends only on $l_{1,1}$ and b_1. This gives value y_1 using only one division in $\mathbb{F}$, namely computing $b_1/l_{1,1}$. For y_2 we compute $(b_2 - l_{2,1}y_1)/l_{2,2} = y_2$ with no unknown on the left hand side. As A is non-singular and therefore L as well, $l_{i,i}$ is never zero and therefore the values y_i can be computed for $i = 1, \ldots, n$.

Before we go to the next step we observe

$$Ax = b \Longrightarrow$$

$$LUx = Ly \text{ as } A = LU \text{ and } b = Ly$$

$$\text{and obtain } Ux = y \text{ by applying } L^{-1}\text{on both sides}$$

It is important to see that we do not need to compute L^{-1} but use the fact that L^{-1} exists for L non-singular and therefore the last equation holds for U, x and y.

We now exploit the fact U is upper triangular. Therefore we can solve $Ux = y$ computationally very efficient manner using the same algorithm as for $Ly = b$. As L was *lower* triangular but U is *upper* triangular, we have to reverse the order of steps, that is, we first compute y_n, then $y_{n-1}, \ldots, y_1$.

A similar argument holds for computing $\det L$ and $\det U$ as [RH96]

$$\det L = l_{1,1} \cdot l_{2,2} \cdot \ldots \cdot l_{n,n} \text{ and } \det U = u_{1,1} \cdot u_{2,2} \cdot \ldots \cdot u_{n,n}$$

So computing $\det A$ can be simplified to $\prod_{i=1}^{n} l_{i,i} u_{i,i}$. As Jama restricts $l_{i,i} = 1$, we save approx. half of our computations and obtain $\det A = \prod_{i=1}^{n} u_{i,i}$.

In classes `LUDecompositionX`, the algorithm outlined above are implemented in different methods. Constructor `LUDecomposistionX(MatrixX A)` computes LU decomposition for given matrix A. Method `det()` computes the determinant of A and method `solve(int[] b, int[] x)` computes the solution x for $Ax = b$ with $b, x \in \mathbb{F}^n$.

As L and U are triangular and therefore approx. half of their coefficients are 0, only one matrix `LU` is used to store all coefficients of *both* matrices. As the diagonal of L is all 1, we do not need to store it and therefore store $u_{i,i}$ on the diagonal of `LU`.

For the sake of speed, Jama introduces a permutation vector `piv`. The idea of `piv` is to replace transposition operations on L and U with a permutation $\pi \in S_n$ of the input, that is, instead of solving

$$L \cdot y = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} \text{ we solve } L \cdot y = \begin{pmatrix} b_{\pi(1)} \\ \vdots \\ b_{\pi(n)} \end{pmatrix}$$

So additional $O(n)$ of space is required to hold permutation $\pi \in S_n$ for an LU decomposition. This space is negligible in comparison to $O(n^2)$ needed to store matrix `LU`.

As [JAMA] was designed to deal with real numbers instead of finite field elements, it was necessary to adapt both the matrix class `MatrixX` and the

LU decomposition class `LUDecompositionX` from JAMA to operations over $\mathbb{F} = \mathrm{GF}(2)$ or $\mathrm{GF}(4)$.

We will show these changed at the code for computing the determinant of matrix A after the LU decomposition of A is done:

```
   /** Determinant
   @return     det(A)
   */
1. public int det () {
2.    Check.assert(m == n, "Matrix must be square");
3.    int d = LU[0][0];
4.    for (int j = 1; j < n; j++)
5.       Field4.mul(d,LU[j][j]);
6.    return d;
7. }
```

As our result is from GF(4) and we use integers to store these field elements, the result of `det()` is `int` (see line 1). Line 2 makes sure that we really deal with a square matrix A as the determinant of A is not defined otherwise. Lines 3-5 compute the product $\prod_{i=1}^{n} u_{i,i}$. Due to the fact that arrays in Java start with 0, we had to rename our variables, so $u_{1,1} \longrightarrow$ `LU[0][0]`, ..., $u_{n,n} \longrightarrow$ `LU[n-1][n-1]`. In addition, as $l_{i,i} = 1$ in Jama, we do not have to take them into account. After we finished the for loop (line 4), we return $\det A =$ `d`.

To solve equation $Ax = b$, Jama uses another trick to save space as this method shows:

```
public void solve (int[] b, int[] x) {
    Check.assert(b.length == m,
      "Matrix row dimensions must be the same as vector size.");
    Check.assert(b.length == x.length,
      "Both vectors must have the same size.");
    Check.assert(Field4.isElement(b), "b is not of Field4");
    Check.assert(isNonsingular(), "Matrix is singular.");

    for (int i = 0; i < b.length; i++)
      x[i] = b[piv[i]];

    for (int k = 0; k < n; k++) {
       for (int i = k+1; i < n; i++) {
          x[i] = Field4.sub(x[i],Field4.mul(x[k],LU[i][k]));
       }
    }

     for (int k = n-1; k >= 0; k--) {
```

```
14.          x[k] = Field4.div(x[k],LU[k][k]);
15.         for (int i = 0; i < k; i++)
16.           x[i] = Field4.sub(x[i],Field4.mul(x[k],LU[i][k]));
17.       }
18.    }
```

Lines 2-5 make sure that all assumptions are satisfied. Lines 6-7 apply the permutation π to b (see above). The equations with lower (lines 8-12) and upper (lines 13-17) triangular matrix are solved below. We notice that Jama computes $b_i = b_i - (y_k \cdot l_{i,k})$ in lines 9-10 and $x_i = x_i - (y_k \cdot u_{i,k})$ in lines 15-16. Using the fact that after computing y_i we do not need b_i any longer, Jama does not need two vectors b and y but only one vector `x` to store the intermediate result for y. A similar argument holds for y and x. Therefore the code outlined above solves $Ax = b$ in place and does not require any memory allocation apart from registers `i,k`. The code above has therefore memory complexity $O(1)$. Inspecting the structure of the loops (lines 8, 9, 13 and 15) we see that we have time complexity $O(n^2)$ for any given matrix $M \in \mathbb{F}^{n\times n}$ and any vector $b \in \mathbb{F}^n$.

6.2.5 Root Finding

As outlined in Section 2.2, the Berlekamp trace algorithm is used for root finding. For applying this algorithm to $f(x) - y$,

$$g(x) := \gcd(f(x) - y, x^q - x) \text{ where } q = |\mathbb{E}|$$

must be computed. As we deal with $\mathbb{E} = \mathrm{GF}(2^n)$, we can compute $x^q = x^{2^n}$ by repeated squaring. Moreover, as we compute the greatest common divisor of $f(x) - y$ and $x^q - x$, we can compute all squares of x modulo $f(x) - y$ [LN86]. This is an upper memory bound and therefore speeds up the computation.

Second, the Berlekamp Trace Algorithm does not find the factorization of a given polynomial $g(x)$ at once but gives factors $a(x)b(x) = g(x)$ after each step with degree $\partial a(x), \partial b(x) \geq 0$. Only if $\partial a(x) \geq 1$ or $\partial b(x) \geq 1$, it yields a non-trivial factorization of $g(x)$. We can distinguish the following cases:

1. $\partial a(x) = 0$ or $\partial b(x) = 0$: In this case, either $b(x) = g(x)$ or $a(x) = g(x)$, so we obtain a trivial factorization,

2. $\partial a(x) = 1$ or $\partial b(x) = 1$: In this case, $a(x)$ or $b(x)$ are a root of $g(x)$ and hence a root of $f(x) - y$,

3. $\partial a(x)$ and $b(x) \geq 2$: Here $g(x)$ splits in non-trivial factors.

Case (1) does not help factoring $g(x)$. However, [LN86] proves that this case can occur during the Berlekamp Trace Algorithm but that there must be at least one Case (2) or (3), that is, the Berlekamp Trace Algorithm will find at least one non-trivial factorization. Case (3) is better but not perfect as it does not give a root of $f(x) - y$. Anyway, applying the algorithm iteratively to $a(x)$ and $b(x)$ will yield all roots of $f(x) - y$ [LN86].

Revisiting this algorithm we see that it is possible (and as a matter of fact, will happen) that we obtain $\partial a(x) = 1$ and $\partial b(x) > 1$, that is, we obtain one root of $f(x) - y$ and one polynomial $b(x) : \partial b(x) > 1$. In the case of a signature algorithm, we are only interested in **one** root of $f(x) - y$, that is, we do not need to compute all roots of $f(x) - y$. In the case of a decryption algorithm, we are only interested in a **certain** root of $f(x) - y$, namely the $x : h(S^{-1} \circ x) = r$, that is, applying affine transformation S^{-1} to x and compute if the hash value of this new x' is equal to precomputed redundancy r.

This observation reflects in Java interface `CheckRoot`:

```
interface CheckRoot {
   public boolean checkRoot(Field2_n root);
}
```

As we see above, it contains of only one function, namely `checkRoot`. The idea of this function is that the root finding class calls it every time Case (2) occurs. When `checkRoot` returns `true`, the correct root was found and the root finding algorithm can return to its caller. Otherwise, it has to continue until the correct one is found in a later step or there are no more roots of $f(x) - y$.

In the case of a signature algorithm, `checkRoot` will always return `true`, in the case of a decryption algorithm, both the inverse transformation S^{-1} of solution `root` and the redundancy of ($S^{-1} \circ$ `root`) have to be computed. If they are equal to redundancy r, the method will return `true` and `false` otherwise. The introduction of the interface `CheckRoot` gives a remarkable speed up for signature and a high speed up for decryption. The reason for implementing it as an interface rather than direct calls between two classes is to allow flexibility (for example, splitting `HFE2_n` to one signature and one decryption class) but to use the same root finding algorithm at the same time.

As we now have all the important bits and pieces together, we can move on to see how public key K is generated using private key k.

6.3 Generation of the Public Key

As outlined in Section 2.3, HFE uses publicly known polynomials $K = (p_1, \ldots, p_n)$ as public key to bypass the private key $k = (S, P, T)$.

In Section 2.3, some steps of the key generation are omitted to make the overall idea clearer. This section will give a more detailed description how keys are generated in HFE.

6.3.1 Bases

First of all, we have to construct a base B for $\mathbb{F}^n$. Generally speaking, any n linear independent vectors $(b_1, \ldots, b_n)$ will do. But to make life easier, we will use the canonical base, that is, $b_i = (\delta_{1,i}, \delta_{2,i} \ldots \delta_{n,i})$ where

$$\delta_{i,j} := \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

is the Kronecker symbol $\delta_{i,j}$. So each $v \in \mathbb{F}^n$ can be written as $v = (v_1, \ldots, v_n)$ with $v_i \in \mathbb{F}$ regarding base B.

We now have to make this vector space compatible with vector space $\mathbb{F}[t]/i(t)$ where $i(t)$ is an irreducible polynomial of degree n. We find that $u(t) \in \mathbb{F}[t]/i(t)$ can be written as

$$u_n t^{n-1} + u_{n-1} t^{n-2} + \cdots + u_2 t + u_1 \text{ with } u_i \in \mathbb{F}$$

If degree $\partial u(t) \geq n$, we can reduce $u(t)$ using $i(t)$ and therefore obtain the form denoted above $\forall u(t) \in \mathbb{F}[t]/i(t)$. For this vector space, we have a canonical base, too. This base consists of the polynomials $q_i(t) = \sum_{j=0}^{n-1} \delta_{i,j} t^j$. Setting $u_i = v_i$ we get for every element in $v \in \mathbb{F}^n$ exactly one associated element $u(t) \in \mathbb{F}[t]/i(t)$ and vice versa. Addition in these two vector spaces follows the same rules, namely componentwise addition of u_i and v_i over $\mathbb{F}$. However, multiplication is different. But as we are mainly interested in the vector space $\mathbb{F}[t]/i(t)$, we will follow the multiplication rules in this vector space.

Having prepared the setting, we can go to our third aim, namely not to talk about single elements $v \in \mathbb{F}^n$ and $u(t) \in \mathbb{F}[t]/i(t)$ but about *all* possible elements from $\mathbb{F}^n$ as this is our message space. Therefore we need to generalize the above bijection $\mathbb{F}^n \longrightarrow \mathbb{F}[t]/i(t)$. We observe that $(p_1, \ldots, p_n)$ with $p_i(x_1, \ldots, x_n) = \sum_{j=1}^{n} a_{i,j} x_i + a_{i,0}$ is an affine transformation for

$$p = M \cdot x + m = \begin{pmatrix} a_{1,1} & \ldots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \ldots & a_{n,n} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} a_{1,0} \\ \vdots \\ a_{1,n} \end{pmatrix}$$

$$\text{with } \det M \neq 0 \text{ and } p = \begin{pmatrix} p_1(x_1, \ldots, x_n) \\ \vdots \\ p_n(x_1, \ldots, x_n) \end{pmatrix}$$

This means we can regard $p_1, \ldots, p_n$ either as n polynomials in n independent variables each or as an affine transformation $\mathbb{F}^n \longrightarrow \mathbb{F}^n$.

Moreover, using this point of view, we can compute u_i for $u(t) \in \mathbb{F}[t]/i(t)$ for each $v \in \mathbb{F}^n$ by replacing $u_i = p_i(x_1, \ldots, x_n)$.

But as we know that the transformation $S = (M \in \mathbb{F}^{n \times n}, n \in \mathbb{F}^n)$ in HFE is affine, we can use the polynomials associated with S as a base for $\mathbb{F}[t]/i(t)$:

$$p_i(x_1, \ldots, x_n) := m_i + m_{i,1}x_1 + m_{i,2}x_2 + \cdots + m_{i,n}x_n$$

We can also see this as a transformation of the canonical base C

$$C := (q_1, \ldots, q_n) \text{ with } q_i(x_1, \ldots, x_n) = x_i$$

As linear independence is invariant under affine transformations, we can use C to see that S is a base for $\mathbb{F}^n$.

Secondly, this base $B := (p_1, \ldots, x_n)$ represents the first step in HFE: transferring message m from the canonical vector space to the vector space generated by S. Therefore using this base has a double advantage for us: First of all it is easy to compute and secondly we have to apply base B to message m to follow the structure of HFE anyway.

6.3.2 Coefficient Transformation

We now have to apply transformation S to each coefficient c_i of P. This can be done very easily by using the `Transform` class and applying it to each coefficient c_i. As the coefficients are values from $\mathbb{E}$, we do not need to compute $p_1, \ldots, p_n$ for this step but can apply S directly to each c_i obtaining c_i^S

We will now inspect how this ides can be expressed in Java:

```
/** applies the transformation to a given vector, that is, res = M*v^T + m
  @param vec vector which should be transformed using this
  @param res vector under this transformation
*/
public void transform(int[] vec, int[] res) {
  Check.assert(Field4.isElement(vec), "vec is not over Field4.");
  Check.assert(vec.length == rowCol,
    "vec has the wrong length, namely " + vec.length);
```

```
4.    Check.assert(res.length == rowCol,
        "res has the wrong length, namely " + vec.length);

5.    M.mul(vec,res);  // apply M to our input
      // add the two vectors
6.    for (int i = 0; i < rowCol; i++)
7.      res[i] = Field4.add(res[i],m[i]);
8.  }
```

The assertion checking facility from class `Check` is used in line 2-4 to find out if all parameters are within the range for an affine transformation. This assertion checker is not used during performance measurements.

In line 5 we apply matrix $M \in \mathbb{F}^{n\times n}$ to our vector `vec`, obtaining vector `res`. Note that for the sake of speed, `res` is a parameter of method `transform` and therefore `transform` has type `void` (see line 1). After this matrix multiplication, we add vector `res` and vector `m` componentwise (lines 6+7) and thereby get our overall result `res`.

6.3.3 Exponentiation in $\mathbb{F}$

After these two steps of precomputation, we have to apply private polynomial P to base B, that is, to compute u_i^h for h_i being the powers of P and u being an element of the vector space $\mathbb{F}[t]/i(t)$.

To apply P to u, we represent u as $u_n t^{n-1} + u_{n-1}t^{n-2} + \cdots + u_2 t + u_1$ with $u_i \in \mathbb{F}$. We compute a power of u as outlined in Figure 6.

$$
\begin{aligned}
u^{q^m} = & \\
& (u_n t^{n-1} + u_{n-1}t^{n-2} + \cdots + u_2 t + u_1)^{q^m} = \\
& (u_n^{q^m} t^{(n-1)q^m} + u_{n-1}^{q^m} t^{(n-2)q^m} + \cdots + u_2^{q^m} t^{q^m} + u_1^{q_m}) = \\
& (u_n t^{(n-1)q^m} + u_{n-1} t^{(n-2)q^m} + \cdots + u_2 t^{q^m} + u_1)
\end{aligned}
$$

Figure 6: Computing u^{q^m} over $\mathbb{F}$

The first step in Figure 6 can be done as q is a multiple of field characteristic p and $(a+b)^{pl} = a^{pl} + b^{pl}$ $\forall a, b \in \mathbb{F}$ and $l \in \mathbb{N}$. The second equation holds as $c^q = c$ in $\mathbb{F}$ $\forall c \in \mathbb{F}$.

This means that we can ignore q^m in terms of $u_i \in \mathbb{F}$ and only have to concentrate on t^{q^m}. We do this by reducing all t^{q^m} by $i(t)$ where $q^m \geq n = \partial i(t)$ whenever we raise $u = (u_1, \ldots, u_n)$ to the power q^m.

The following example (see Figure 7) with $\mathbb{F} = \mathrm{GF}(4)$, generated by $j(s) = s^2 + s + 1$ with $s \in \mathrm{GF}(2)$, will illustrate this principle. As we have to represent our elements from GF(4) in a computer, we will use the notation $\mathrm{GF}(4) = \{0, 1, 2, 3\}$ for the elements of $\mathbb{F}$. That might be a little bit annoying from a mathematical point of view. We are motivated by the desire to have a representation that is humanly readable and mathematically correct but also a representation that can be used in a computer fairly easily. The easiest way to do this it to represent these four elements of GF(4) as integers. So elements of GF(4) can be seen as:

- additive neutral: 0
  (that is, $(0, 0)$ in terms of $\mathrm{GF}(2) \times \mathrm{GF}(2)$ and 0 in terms of $\mathbb{F}_2[s]/j(s)$)
- multiplicative neutral: 1 (that is, $(0, 1)$ and 1)
- generator γ: 2 (that is, $(1, 0)$ and x in our representation)
- γ^2: 3 (that is, $(1, 1)$ and $x + 1$)

Addition and multiplication follows the usual rules in GF(4), that is, they are done modulo the generating polynomial $j(s)$.

In this example (see Figure 7) we will compute $u^4 = u^q$ for $u \in \mathbb{E} = \mathrm{GF}(64)$ with generating polynomial $i(t) = t^3 + t^2 + 2t + 3$ for $\mathbb{E}$ and $t \in \mathbb{F}$. To reduce higher degrees of t, we can use the reduction table from Figure 8.

$$
\begin{aligned}
u^4 \mathrel{\hat{=}}& \\
&(u_3 t^2 + u_2 t + u_1)^4 = \\
&u_3 t^8 + u_2 t^4 + u_1 = \\
&u_3(2t + 3) + u_2(3t^2 + t + 3) + u_1 = \\
&(2tu_3 + 3u_3) + (3t^2 u_2 + tu_2 + 3u_2) + u_1 = \\
&(3u_2)t^2 + (2u_3 + u_2)t + (3u_3 + 3u_2 + u_1)
\end{aligned}
$$

Figure 7: u^4 with $u \in \mathrm{GF}(64)$ over GF(4)

A similar argument holds when we compute $u^{q^k + q^l}$ with $k, l \in \mathbb{N}$ (see Figure 9).

Multiplying out the last equation in Figure 9 gives terms $u_i u_j$, $1 \le i, j < n$, that is, quadratic terms in $u_1, \ldots, u_n$. This is different to Figure 6 where all terms where linear in u_i.

$$\begin{array}{rl} t^3 \to & t^2+2t+3 \\ t^4 \to & t^3+2t^2+3t \to \\ & 3t^2+t+3 \\ t^8 \to & t^7+2t^6+3t^5 \to \\ & 2t+3 \end{array}$$

Figure 8: Reduction Table for $i(t) = t^2 + 2t + 3$

$$\begin{aligned} u^{q^k+q^l} &= \\ &= (u_n t^{n-1} + u_{n-1} t^{n-2} + \cdots + u_2 t + u_1)^{q^k+q^l} = \\ &= (u_n t^{n-1} + u_{n-1} t^{n-2} + \cdots + u_2 t + u_1)^{q^k} \\ &\quad (u_n t^{n-1} + u_{n-1} t^{n-2} + \cdots + u_2 t + u_1)^{q^l} \\ &= (u_n^{q^k} t^{(n-1)(q^k)} + u_{n-1}^{q^k} t^{(n-2)(q^k)} + \cdots + u_2^{q^k} t^{q^k} + u_1^{q^k}) \\ &\quad (u_n^{q^l} t^{(n-1)(q^l)} + u_{n-1}^{q^l} t^{(n-2)(q^l)} + \cdots + u_2^{q^l} t^{q^l} + u_1^{q^l}) \\ &= (u_n t^{(n-1)(q^k)} + u_{n-1} t^{(n-2)(q^k)} + \cdots + u_2 t^{q^k} + u_1) \\ &\quad (u_n t^{(n-1)(q^l)} + u_{n-1} t^{(n-2)(q^l)} + \cdots + u_2 t^{q^l} + u_1) \end{aligned}$$

Figure 9: $u^{q^k+q^l}$ in $\mathbb{F}[t]/i(t)$

As $i(t)$ cannot affect u_i, the only thing left to do in Figure 9 is to compute the reduction of the powers t^m with $m = (n-1)(q^l+q^l), (n-2)(q^l+q^l), \ldots, (q^l+q^l)$ using a reduction table similar to Figure 8.

We will illustrate this in the same $\mathbb{F}$ and $\mathbb{E}$ as above but with $u^5 = u^{4+1} = u^{q^1+q^0}$ in Figure 10.

Terms in Figure 10 can now be reduced using $i(t)$ or a reduction table as in Figure 8. Please note that this does not affect u_i but only t and therefore does not spoil the overall structure of u^{4+1}, that is, the fact that each term depends on exactly two u_i, u_j.

Having these observations, we have now to take into account that our message m is first transformed by S, that is, we change the underlying vector space. Therefore we compute base $B = (p_1, \ldots, p_n)$ as described in Section 6.3.1. To combine the fact that we deal with an isomorphic vector space rather than $\mathbb{E}$, we replace each u_i with the corresponding polynomial

$$
\begin{aligned}
u^{4+1} &\mathrel{\hat{=}} \\
&\mathrel{\hat{=}} (u_3t^2 + u_2t + u_1)^{4+1} \\
&= (u_3t^2 + u_2t + u_1)^4 (u_3t^2 + u_2t + u_1)^1 \\
&= (u_3t^8 + u_2t^4 + u_1)(u_3t^2 + u_2t + u_1) \\
&= (u_3t^8 + u_2t^4 + u_1)(u_3t^2 + u_2t + u_1) \\
&= u_3^2t^{10} + u_2u_3t^9 + u_1u_3t^8 + u_2u_3t^6 + u_2^2t^5 + u_1u_2t^4 + u_1u_3t^2 + u_1u_2t + u_1^2
\end{aligned}
$$

Figure 10: u^5 with $u \in \mathrm{GF}(64)$ over $\mathrm{GF}(4)$

$p_i(x_1, \ldots, x_n)$ (see Section 6.3.1). This means that all our operations are now defined not on $u_i \in \mathbb{F}$ but on polynomials p_i over $\mathbb{F}$.

For our above example, we have to define transformation $S = (M, m)$ and base B. We use:

$$
M := \begin{pmatrix} 0 & 1 & 2 \\ 3 & 0 & 1 \\ 2 & 3 & 1 \end{pmatrix} \text{ and } m := \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \text{ as } \det(M) = 3 \text{ over GF}(4)
$$

$$
\text{This gives base } B := \left\{ \begin{array}{lllll} p_1(x_1,x_2,x_3) := & & x_2 & +2x_3 & +1, \\ p_2(x_1,x_2,x_3) := & 3x_1 & & +x_3 & +2, \\ p_3(x_1,x_2,x_3) := & 2x_1 & +3x_2 & +x_3 & +3 \end{array} \right\}
$$

Using $\mathbb{F}$ and $\mathbb{E}$ as defined above, we now compute x^4 in terms of the vector space generated by S:

$$
\begin{aligned}
x^4 \mathrel{\hat{=}} u^4 \quad & \mathrel{\hat{=}} \\
& \mathrel{\hat{=}} (3p_2)t^2 + (2p_3 + p_2)t + (3p_3 + 3p_2 + p_1) \\
& = 3(3x_1 + x_3 + 2)t^2 + \\
& \quad [2(2x_1 + 3x_2 + x_3 + 3) + (3x_1 + x_3 + 2)]t + \\
& \quad [3(2x_1 + 3x_2 + x_3 + 3) + 3(3x_1 + x_3 + 2) + (x_2 + 2x_3 + 1)] \\
& = (2x_1 + 3x_3 + 1)t^2 \\
& \quad [(3x_1 + x_2 + 3x_3 + 2) + (3x_1 + x_3 + 2)]t + \\
& \quad [(x_1 + 2x_2 + 3x_3 + 2) + (2x_1 + 3x_3 + 1) + (x_2 + 2x_3 + 1)] \\
& = (2x_1 + 3x_3 + 1)t^2 \\
& \quad [(3+3)x_1 + x_2 + (3+1)x_3 + (2+2)]t + \\
& \quad [(1+2)x_1 + (2+1)x_2 + (3+3+2)x_3 + (2+1+1)] \\
& = (2x_1 + 3x_3 + 1)t^2 \\
& \quad [x_2 + 2x_3 +]t + \\
& \quad [3x_1 + 3x_2 + 2x_3 + 2]
\end{aligned}
$$

As we observe, all terms in the last equation depend on x_1, x_2, x_3 and also t. They are linear. As B is a base, we could express all terms with respect to this base but this would lead to the first line of the equation above and is therefore of no use for our purpose.

The technique used in the above example can also be used in the general case and leads to linear equations in the case x^{q^m} and quadratic equations in the case x^{q^k+q+l}, regarding $x_1, \ldots, x_n$.

6.3.4 Multiplying the constants

The last bit we need before we can apply P to base B are the constants c_i. In Section 6.3.2 we saw that we have to transfer each c_i using S as we now work in another vector space. These $c_i \in \mathbb{E}$ can each be represented over $\mathbb{F}[t]/i(t)$ as

$$c_i = (c_{i,1}, \ldots, c_{i,n}) = c_{i,n}t^{n-1} + c_{i,n-1}t^{n-2} + \ldots + c_{i,2}t + c_{i,1} \text{ with } c_{i,j} \in \mathbb{F}$$

and are therefore compatible with our representation of u^{q^m} and $u^{q^k+q^l}$ in terms of base B. As u^{q^m}, $u^{q^k+q^l}$ and c_i depend on multiple powers of t, we have to reduce by $i(t)$ after multiplying $c_i \cdot u^{q^m}$, $c_i \cdot u^{+q^l}$.

As c_i is independent of $x_1, \ldots, x_n$, we are sure that it does not change the degree of our polynomials $p_1, \ldots, p_n$ and therefore $c_i \cdot u^{q^m}$ can still be expressed with a linear polynomial in terms of $x_1, \ldots, x_n$ and $c_i \cdot u^{q^k + q^l}$ with a quadratic polynomial in terms of $x_1, \ldots, x_n$.

6.3.5 Applying T

Applying T to $p_1, \ldots, p_n$ seems to be quite tricky as we do not apply an affine transformation to an element of $\mathbb{F}$ but to polynomials. But this is only true at first glance. When we take a second look at the problem we can use the replacement technique discussed in Sections 6.3.1 and 6.3.3. This means we first compute

$$T : M \cdot u + m = \begin{pmatrix} m_{1,1} & \ldots & m_{1,n} \\ \vdots & \ddots & \vdots \\ m_{n,1} & \ldots & m_{n,n} \end{pmatrix} \cdot \begin{pmatrix} u_1 \\ \vdots \\ u_n \end{pmatrix} + \begin{pmatrix} m_1 \\ \vdots \\ m_n \end{pmatrix}$$

and then replace u_i with the corresponding polynomial p_i. After this step we can collect all x_i and $x_i x_j$ for each row in the output vector and obtain new polynomials

$$\begin{aligned} p_1(x_1, \ldots, x_n) &= a_{1,0} + a_{1,1}x_1 + \cdots + a_{1,n}x_n + \\ &\quad a_{1,1,1}x_1^2 + a_{1,1,2}x_1x_2 + \cdots + a_{1,n,n}x_n^2 \\ &\vdots \\ p_n(x_1, \ldots, x_n) &= a_{n,0} + a_{n,1}x_1 + \cdots + a_{n,n}x_n + \\ &\quad a_{n,1,1}x_1^2 + a_{n,1,2}x_1x_2 + \cdots + a_{n,n,n}x_n^2 \end{aligned}$$

As the transformation T does not affect x_i and $x_i x_j$, these polynomials are of degree 2 at most.

We will inspect some Java code from class `Transform4` to see how these ideas are expressed in the HFE implementation:

```
public Poly4MultiSquare[] transform(Poly4MultiSquare[] in) {
  Check.assert(in.length == rowCol,
    "Wrong number of polynomials.");
  final int terms = Poly4MultiSquare.tau(rowCol);

  int[][] Ms = M.getArray();
  int[][] polys = new int[rowCol][];
  for (int i = 0; i < rowCol; i++)
```

```
 7.     polys[i] = in[i].coeffs();

 8.   Poly4MultiSquare[] res =
        new Poly4MultiSquare[rowCol];
 9.   for (int i = 0; i < rowCol; i++) {
10.     int[] a = new int[terms];
11.     for (int j = 0; j < terms; j++) {
12.       int sum = (j == 0) ? m[i] : 0;
13.       for (int k = 0; k < rowCol; k++) {
14.         sum = Field4.add(sum,Field4.mul(
                    polys[k][j],Ms[i][k]));
15.       }
16.       a[j] = sum;
17.     }
18.     res[i] = new Poly4MultiSquare(rowCol,a);
19.   }
20.   return res;
21. }
```

In line 2, the assertion checker from class `Check` is used to ensure that input `in` and object variable `rowCol` have the same size, that is, our input consists of as many quadratic polynomials as matrix $M \in \mathbb{F}^{n\times n}$ has rows. In line 3, the function $\tau(n)$ from Section 6.2.3 is used to compute the number of terms we expect for input `in` and need to compute for output `res`. Lines 4-7 are here for speed purposes only and give faster access to coefficients from matrix `M` and polynomials `polys`. Note that these operations do not copy the whole array but only a pointer to these arrays. In line 8 memory is allocated for result `res`.

The core algorithm is in lines 9-19. The first `for` loop (line 9) is over the result polynomial `poly[i]`, the second `for` loop (line 11) is over the coefficients $0, \ldots, \tau(n) - 1$ for this polynomial `poly[i]` and finally, the third `for` loop (line 13) iterates over the input polynomials `polys`.

Therefore, in line 14, we multiply matrix `M` with each corresponding coefficient in polynomials `polys`. Summing up these products in terms of GF(4), we obtain coefficient `j` for each output polynomial `res[i]`. The coefficients are numbered as described in Section 6.2.3, that is, 0 is the constant term, $1, \ldots, n$ are the linear terms and $(n+1), \ldots, \tau(n) - 1$ are the quadratic terms regarding x_i. For the constant term (that is, `a[0]`) of each polynomial, we not only have to take into account input polynomials `polys` but also vector `m` of affine transformation T. This is done in line 12 by checking the condition `j == 0`.

Last but not least, line 18 generates the polynomials `res[i]` using the coefficients `a[0..`τ`(n)-1]`. The constructor `Poly4MultiSquare(rowCol,a)` does not allocate new memory nor copies `a` for the sake of speed. Therefore we have to allocate new memory for `a` in each iteration. We can see this in line 10.

6.3.6 Overall Algorithm

As the computations outlined above are done in different classes, our overall algorithm for key generation is fairly simple:

1. Compute base B.
2. Compute u^{q^i} for $i = 0, 1, \ldots a$ with $q^a < d < q^{a+1}$.
3. Compute $u^{q^k+q^l} = u^{q^k} u^{q^l}$ using the results of step 2.
4. Compute $c_i u^{h_i}$ using the results from step 2 and 3.
5. Add up the results of step 4.
6. Apply T to the result of step 5.

The result of this algorithm is a vector of n polynomials in n variables each.

6.3.7 Acceleration

As we saw in Section 6.3.3, we only get linear polynomials in step 2 of the algorithms described above. Therefore we have a special class `PolyXMultiLinear`, which does all operations in $O(n)$. This is a remarkable speed up for key generation. Class `PolyXMultiSquare` is used in steps 3-6 as the polynomials obtained are quadratic.

For our next acceleration we observe that $q^{i+1} = q^i q$, that is, $u^{q^{i+1}} = u^{q^i q} = (u^{q^i})^q$ and therefore computing $u^{q^{i+1}}$ can make use of the previous value u^{q^i}. Second we see that we can apply our reduction step at any time as

$$(a(t) \cdot b(t) \cdot c(t)) \pmod{i(t)} \equiv ((a(t) \cdot b(t)) \pmod{i(t)} \cdot c(t)) \pmod{i(t)}.$$

Therefore we first compute u^{q^i}, apply reduction by $i(t)$ and then compute $(u^{q^i})^q$. Doing this iteratively, we can speed up step 2 of our algorithm.

As we can do reduction after *every* step, we can use this to bound the degree of t both in step 3 and 4 and therefore quicken these two steps, too. As the degree of t is bounded, we also save memory by computing the reduction after every step.

Last but not least we can reduce reduction to a simple copy operation, using a reduction table as shown in Figure 8.

6.4 Speed Measurement

To see how fast this implementation is, several speed measurements were done. All timing results where obtained using a Pentium-III-730MHz running Java 1.3. Moreover, all classes as final were declared to be final and frequently used variables were global static rather than local.

Toy Version

Extension Field	Key Generation	Encryption	Decryption
GF(64)	210 μs	14 μs	56 μs

GF(2^{67}) and GF(2^{129})

In general, timing results are based on 101 independent measurements. This is not true for encryption and signature verification. As these operations were faster than the timer used, it was necessary to group 100 operations together. So these timing results are based on the 10,100 measurements and 101 means were computed from these results. In each case, the tables give the median of 101 values for each entry.

Key Generation

$\mathbb{E} = \mathrm{GF}(2^{67})$	$k = 0$	4453 ms
$\mathbb{E} = \mathrm{GF}(2^{67})$	$k = 8$	4478 ms
$\mathbb{E} = \mathrm{GF}(2^{129})$	$k = 0$	63,341 ms

The time for generating a key with 8 equations replaced by random equations is less than 1 % more than for no equation replaced. Therefore, the influence of parameter k on key generation is negligible. Increasing the size of the extension field has markable influence on time required for key generation.

Encryption and Decryption

		Encryption	Decryption
$\mathbb{E} = \mathrm{GF}(2^{67})$	$k = 0$	4.83 ms	290 ms
$\mathbb{E} = \mathrm{GF}(2^{67})$	$k = 1$	4.82 ms	320 ms
$\mathbb{E} = \mathrm{GF}(2^{67})$	$k = 2$	4.82 ms	844 ms
$\mathbb{E} = \mathrm{GF}(2^{67})$	$k = 4$	4.82 ms	2,527 ms
$\mathbb{E} = \mathrm{GF}(2^{67})$	$k = 8$	4.81 ms	45,698 ms
$\mathbb{E} = \mathrm{GF}(2^{129})$	$k = 0$	28.98 ms	1,337 ms
$\mathbb{E} = \mathrm{GF}(2^{129})$	$k = 1$	29.08 ms	2,658 ms
$\mathbb{E} = \mathrm{GF}(2^{129})$	$k = 2$	29.11 ms	3,986 ms
$\mathbb{E} = \mathrm{GF}(2^{129})$	$k = 4$	29.10 ms	12,023 ms

As expected, time for decryption increases dramatically with the number of equations removed while time for encryption is more or less the same. With approx. 12 seconds for $\mathbb{E} = \mathrm{GF}(2^{129})$, $k = 4$, this Java implementation of HFE- is too slow to be useful for encryption of session keys.

Signature

	Generation	Verification
$\mathbb{E} = \mathrm{GF}(2^{67})$	305 ms	2.47 ms
$\mathbb{E} = \mathrm{GF}(2^{129})$	1,331 ms	20.73 ms

As signature generation is not affected by the number of equations removed (see Section 2.4), there was only one set of measurements for $n = 67, 129$. As expected, the time for signature generation is much lower for $n = 67$ that for $n = 129$. With approx. 1.3 seconds for signature generation, this Java implementation is quite fast. As Flash and SFlash have a different structure for the trap door polynomial used, they can not be compared with HFE-. Quartz, however, is an HFEv- scheme (see Section 4.5). According to [MG01, QFS], Quartz needs approx. 30 seconds to generate one signature. As one Quartz signature uses 4 HFEv- schemes, this can be translated to approx. 7 seconds for one signature in HFE- in the Quartz implementation submitted to [NESSIE].

7 Conclusions and Outlook

This project was about studying and understanding the public key system HFE. This aim was achieved by implementing a Java version of HFE-. However, as the project was planed in October, the stress was more on understanding HFE and less on writing finite field operations in Java. Unfortunately, there is nothing like Victor Shoup's Number Theory Library [NTL] for Java, hence I had to implement the finite field operations myself as porting whole NTL (or related packages, see [NELIB]) was certainly much more than I could do within this project. This gave me a nice inside view of finite fields and their arithmetic and was certainly very useful for my overall understanding of HFE. On the other hand, it slowed down the whole implementation process as I was more concerned about finite field operations than implementing HFE in Java. This changed the overall structure of this project.

When I started this project, I was convinced that HFE can be used for encryption with nearly the same timing results as for signature. As we have seen in Section 6.4, I was proven wrong. While signature and encryption is quite fast, decryption slows down so much that it is far from being usable. This is bad news for public key cryptography as there is a need for encryption schemes but not so much for signature schemes [HFE96]. However, as we have seen in the same section, particular variations for HFE (especially with a small number k of public equations removed) can still be useful for an encryption scheme. Moreover, this Java version is not slower than the C++ implementation of Quartz submitted to [NESSIE]. As the authors of this C++ implementation point out in [QFS], that their implementation is far from being efficient and can be improved. As this Java implementation is faster than their C++ version, I would like to stress their statement.

The most difficult part during this implementation was the public key generation. This is mentioned in only two lines (!) in each of the basic papers [HFE96, QFS]. It took me some time both to understand the overall process and to implement it in Java. On the other hand, this gave me the opportunity to find some tricks and speed ups for the whole process, for example the distinction of linear and quadratic polynomials for the public key generation.

A very interesting part of this project was that it dealt both with the mathematical and the computer science aspects of Hidden Field Equations. As both subjects have different points of view on the same problem, this enriched the project and gave it a much wider view on HFE than a mathematical or computer science project alone would have provied.

However, in my implementation are several things which could improved either for the sake of usability or speed.

Speed

- Introduce faster matrix multiplication (e.g. Strassen [CLR96, pp. 739-745]) instead of the standard method.
- Optimize the finite field operations for the extension fields used in HFE. This can be done by porting the relevant parts of an existing library to Java.
- Optimize the public key operation. One way would be to group 32/64 public polynomials (depending on the word size of the underlying processor) together and apply all operations not on one polynomial but on 32/64 each time.

Usability For usability it would be worthwhile to fit HFE into the existing Java Cryptographic Extension [JCE14]. As the structure of HFE follows the structure for JCE public key cryptographic extensions (e.g. key generation, encryption, decryption), this task is rather identifying the corresponding structures in this HFE implementation and JCE than writing new code.

As we have seen in this report, HFE can be used both for encryption and signature. Although it is currently used only for digital signature (see 4.5), basic HFE can be modified in a way that it withstands all known attacks (see Section 4).

However, the big public key size can be a problem. The McEliece scheme (see section 1.4) has a similar key size and was therefore discarded for practical use [HAC96]. On the other hand, times have changed and memory is first of all cheap in nowadays and second also available on smart cards. An advantage of HFE is moreover the very short signature length (see Section 4.5).

As an overall result I would like to conclude that HFE can be implemented entirely in Java. As this implementation used standard Java in version 1.3 from Sun, I did not use optimized Java compilers or native C++ code for speed critical parts of HFE. The project gave me a nice inside view of HFE and also allowed me to understand finite fields much better.

8 Acknowledgments

I would like to finish this report with acknowledgments to Emanuel Popovici. He gave me important hints how to implement finite field operations.

References

[AM93] Menezes, Alfred J. et. al.: *Applications of finite fields*, Kluwer Academic Publishers Group, 1993. ISBN 0-7923-9282-5

[BSS99] Blake, Ian; Seroussi, Gadiel; Smart Nigel: *Elliptic Curves in Cryptography*, Cambridge University Press, 1999. ISBN 0-521-65374-6

[CLR96] Thomas Cornmen, Charles E. Leiserson, Ronald L. Rivest: *Introduction to Algorithms*, 17$^{\text{th}}$ printing, The MIT Press, McGraw-Hill Book Company, 1996. ISBN 0-262-03141-8 or 0-07-013143-0

[DH76] Whitfield Diffie, Martin Hellman: *New Directions in Cryptography*, IEEE Transactions on Information Theory, 22 (1976), 644-654.

[DK97] Knuth, Donald Ervin: *The art of computer programming - Volume 2: Seminumerical algorithms*, Addison Wesley Longman, 1997. ISBN 0-201-89684-2

[DW98] Wallace, David A.R.: *Groups, Rings and Fields*, Springer-Verlag London Limited, 1998. ISBN 3-540-76177-2

[GJ79] Garay, Michael R. and Johnson, David S.: *Computers and Intractability - A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979. ISBN 0-7167-1044-7 or 0-7167-1045-5

[HAC96] Menezes, Alfred J. et. al.: *Handbook of Applied Cryptography*, CRC Press, 1996, ISBN 0-8493-8523-7

[HFE] *Hidden Field Equations public key cryptosystem home page (HFE)* http://www.minrank.org/hfe/, 5.2.2002

[HFE96] Patarin, Jaques: *Hidden Field Equations (HFE) and Isomorphisms of Polynomials (IP): two new Families of Asymmetric Algorithms - Extended Version* -, http://www.minrank.org/hfe.pdf

[HFE01] Courtois, Nicolas T.: *The security of Hidden Field Equations (HFE)*, http://www.minrank.org/hfesec.pdf

[J14N] Sun Microsystems Inc.: *JavaTM 2 SDK, Standard Edition, version 1.4 - Summary of New Features and Enhancements*, http://java.sun.com/j2se/1.4/docs/relnotes/features.html

[JAAS] Sun Microsystems Inc.: *Assertion Facility*, http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html

[JAMA] The MathWorks, Inc. and the National Institute of Standards and Technology: *JAMA: A Java Matrix Package, Version 1.0.1*, http://math.nist.gov/javanumerics/jama/

[JCE14] *How to Implement a Provider for the Java Cryptography Extension in the Java 2 SDK, Standard Edition, v 1.4*, http://java.sun.com/j2se/1.4/docs/guide /security/jce/HowToImplAJCEProvider.html

[JDOC] Sun Microsystems Inc.: *javadoc - The Java API Documentation Generator* http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/javadoc.html

[JNI] Beth Stearns: *Trail: Java Native Interface* http://java.sun.com/docs/books/tutorial/native1.1/

[KB99] Beck, Kent: *Extreme programming explained: embarace change*, Addison Wesley Longman, 1999 (second printing), ISBN 0-201-61641-6

[KH90] Huber, Klaus: *Some Comments on Zech's Logarithms*, IEEE Transactions on Information Theory, Vol. 36., No. 4, July 1990, pp. 946-950

[LD00] López, Julio and Dahab, Ricardo: *An Overview of Elliptic Curve Cryptography*, http://citeseer.nj.nec.com/333066.html or http://www.dcc.unicamp.br/ic-tr-ftp/2000/00-14.ps.gz

[LN86] Lidl, Rudolf and Niederreiter, Harald: *Introduction to finite fields and their applications*, Cambridge University Press, 1986. ISBN 0-521-30706-6

[MG01] Martinet, Gwenaëlle: *Quartz, Flash and SFlash.* Report for NESSIE, Document NES\DOC\ENS\WP3\006\2, 7th of March 2001, http://www.cosic.esat.kuleuven.ac.be/nessie/reports/enswp3-006-2.pdf

[MI88] T.Matsumoto and H. Imai, *Public Quadratic Polynomial-tupels for efficient signature-verification and message-encryption*, Eurocrypt'88, Springer Verlag 1988, pp.419-453

[NELIB] NESSIE: *Multi-Precision Libraries for submissions for the NESSIE project* http://www.cosic.esat.kuleuven.ac.be/nessie/call/mplibs.html

[NESSIE] *NESSIE - New European Schemes for Signatures, Integrity, and Encryption, IST-1999-12324*, https://www.cosic.esat.kuleuven.ac.be/nessie/

[NTL] Victor Shoup: *NTL: A Library for doing Number Theory* http://www.shoup.net/ntl/

[QFS] Jaques Patarin, Nicolas Courtois and Louis Goublin: *Submission of Quartz, Flash, and SFlash for NESSIE.* http://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions/quartz.zip http://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions/flash.zip http://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions/sflash.zip

[RH96] Hill, Richard O.: *Elementary Linear Algebra with Applications*, Harcourt College Publishers, 1996. ISBN 0-03-010347-9

[ShC01] Shantz, Sheueling Chang: *From Euclid's GCD to Montgomery Multiplication to the Great Divide*, Sun Microsystems, SML Technical Report, 2001, SMLI TR-2001-95. http://research.sun.com/research/techrep/2001/abstract-95.html

[Sh97] Shor, P.: *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, SIAM Journal on Computing, 26(5):1484–1509, 1997.

[SK99] Shamir, Adi and Kipnis, Aviad: *Cryptanalysis of the HFE Public Key Cryptosystem*, Crypto'99.