HFE in Java: Implementing Hidden Field Equations for Public Key Cryptography

Christopher Wolf[□], **Patrick Fitzpatrick**[§], **Simon N. Foley**[†], **and Emanuel Popovici**[⅄]

[□,§] *Department of Mathematics, University College Cork, IRELAND*

[†] *Department of Computer Science, University College Cork, IRELAND*

[⅄] *Department of Electrical Engineering, University College Cork, IRELAND*

E-mail: [□]`chris@Christopher-Wolf.de` [§]`p.fitzpatrick@ucc.ie` [†]`s.foley@cs.ucc.ie` [⅄]`emanuel@rennes.ucc.ie`

Abstract — **Hidden Field Equations (HFE) is a public key system that can be used both for digital signature and encryption. It is based on polynomial equations over finite fields and their extension fields. A version of HFE is under consideration as a European signature standard. In this paper we give a brief description of HFE and an efficient Java implementation. We will also outline some of the advantages and disadvantages of using HFE in practice.**

Keywords — **Public Key Cryptography, Java, Hidden Field Equations, HFE, Public Key Encryption, Public Key Signature**

I Introduction

Public key cryptography is used in e-commerce systems for authentication (electronic signatures) and secure communication (encryption). In contrast to secrete cryptography, public key cryptography has advantages in terms of key distribution. Moreover, signature schemes can not be obtained by secrete key schemes. The security of using current public key cryptography for encryption centres on the difficulty of solving certain classes of problem. The RSA scheme relies on the difficulty of factoring very large numbers, while the difficulty of solving discrete logarithms provide the basis for the ElGamal and Elliptic Curve schemes [1]. Given that the security of these public key schemes rely on such a small number of problems that are *currently* considered hard, research on new schemes that are based on other classes of problems is worthwhile. Such work provides a greater diversity that forces cryptanalysts to expend additional effort by concentrating on a completely new type of problem.

Important results on the potential weaknesses of existing public key schemes are emerging. Techniques for factorization and solving discrete logarithm continually improve. Polynomial time quantum algorithms [2] can be used to solve both problems; fortunately, quantum computers with more than 5 bits are not yet available and it seems unlikely that they will be available within the next couple of decades. However, this does point to the importance of research into new algorithms for asymmetric encryption.

In 1996, Patarin proposed the use of a special class of polynomials over finite fields for public key cryptography [3, 4]. The scheme supports both encryption and digital signatures; its security is related to the difficulty of solving a random selected system of multivariate quadratic equations over GF(2). The scheme improved the Matasumoto-Imai-System [5] that was broken in [6].

This paper outlines some of the design issues for an efficient Java implementation of HFE. Section II provides a short introduction to HFE and the underlying mathematics. Section III describes some of the implementation considerations and Section IV provides preliminary performance results. The conclusion considers advantages and disadvantages of HFE and argues why it is a good choice for certain application domains.

II Introduction to HFE

HFE is based on polynomials over finite fields and extension fields. The general idea is to use a polynomial over an extension field as a private key and a vector of polynomials over the underlying finite field as public key. HFE also uses private affine transformations to hide the extension field and the private polynomial.

a) Mathematical Background

Figures 1 and 2 give an outline of the structure of HFE. S and T represent two affine transformations and P is the private polynomial. The private key is represented by triple (S, P, T).

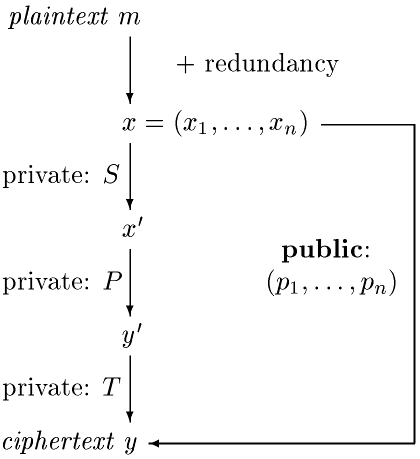

Fig. 1: Message Encryption with HFE

The polynomials $(p_1, \dots, p_n)$ are the public key. These public polynomials as well as the private affine transformations S and T are over $\mathbb{F}$, a finite field with cardinality $q = |\mathbb{F}|$. The private polynomial P is over $\mathbb{E}$, an extension field of $\mathbb{F}$ generated by the irreducible polynomial $i(x)$ of degree n.

Encryption and Decryption of Messages using Private Key

The private polynomial P (with degree d) over $\mathbb{E}$ is an element of $\mathbb{E}[x]$. To keep the public polynomials small, the private polynomial P must have the property that its terms are at most quadratic over $\mathbb{F}$. In the case of $\mathbb{E} = \mathrm{GF}(2^n)$ this means that the powers have Hamming weight at most 2. In symbols:

$$\begin{aligned} P &: \mathbb{E} \to \mathbb{E} \\ P(x) &= \sum c_i x^{h_i}, \text{where} \\ & c_i \in \mathbb{E}, h_i \le d, h_i \ne h_j \ \forall i \ne j, \\ & h_i = \begin{cases} 0, & \text{(constant term)} \\ q^a, & a \in \mathbb{N}_0 \\ & \text{(linear factors)} \\ q^b + q^c, & b, c \in \mathbb{N}_0 \\ & \text{(quadratic factors)} \end{cases} \end{aligned}$$

Since the affine transformations S and T are over $\mathbb{F}$ it is necessary to transfer the message m from $\mathbb{E}$ to $\mathbb{F}^n$ in order to encrypt it. This is done by regarding m as a vector $(x_1, \dots, x_n) \in \mathbb{F}^n$. Thus we no longer think about the extension field as a field but as an n-dimensional vectorspace over $\mathbb{F}$ with the rows of the identity matrix I as base of $\mathbb{F}^n$. To encrypt $(x_1, \dots, x_n)$ we first apply S, giving the result denoted x'. At this point x' is transferred from $\mathbb{F}^n$ to $\mathbb{E}$ so we can apply the private polynomial P which is over $\mathbb{E}$. The result is denoted as $y' \in \mathbb{E}$. Once again, y' is transferred to the vector $(y'_1, \dots, y'_n)$, the transformation T is applied and the final output $y \in \mathbb{E}$ is produced from $(y_1, \dots, y_n) \in \mathbb{F}^n$.

To decrypt y, the above steps are done in reverse order. This is possible if the private key (S, P, T) is known. The crucial step in the deciphering is not the inversion of S and T, but rather the computation of the solutions of $f(x') = y'$. As P has degree d there are up to d different solutions for this equation. Addition of redundancy to the message m provides an error-correcting effect that 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).

Public Key: Generation and Encryption

As encryption must be possible without the secret key (S, P, T) we therefore compute publicly known polynomials $(p_1, \dots, p_n)$ which apply (S, P, T) to our message m without revealing the private key. To compute the public polynomials, we now trans-

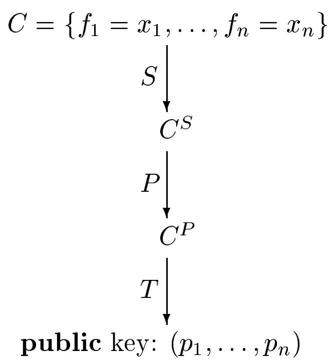

Fig. 2: Public Key Generation in HFE

fer P from $\mathbb{E}$ to $\mathbb{F}^n$. Thus we apply the same steps as above to the canonical base of polynomials

$$\begin{aligned} C = \quad & \{f_1(x_1, \dots, x_n) = x_1, \\ & \qquad \vdots \\ & f_n(x_1, \dots, x_n) = x_n\} \end{aligned}$$

for $\mathbb{E}$ over $\mathbb{F}$ and think of x as a vector with n independent variables, namely $x_1, \dots, x_n$. The same is true for the result of $P(x)$ which also can be represented as a vector. Given this perspective, we no longer have one polynomial in one variable, but n polynomials in n independent variables, namely

$p_1(x_1, \ldots, x_n), \ldots, p_n(x_1, \ldots, x_n)$. For the remainder of this section, we no longer work over $\mathbb{E}$, but over $\mathbb{F}[x]/i(x)$, using the normal rules for addition and multiplication of polynomials modulo $i(x)$.

Figure 2 outlines how the public key is generated from the canonical base C. Initially, the affine transformation S is applied to the base C of $\mathbb{F}^n$, giving C^S. This step hides the underlying field to an attacker. Next, we can compute the public polynomials using the private polynomial P. We first transfer all of its coefficients c_i using S, giving results c_i^S. Second, we raise C^S to the powers h_i of our private polynomial $P(x)$, multiply this by c_i^S each, add the results, and denote this by C^P. Finally we also hide C^P by applying T. This completes the computation of our public polynomials $(p_1(x_1, \ldots, x_n), \ldots, p_n(x_1, \ldots, x_n))$.

The public polynomials $p_1, \ldots, p_n$ are each of degree at most 2 as the powers h_i of P are chosen carefully to ensure this property (see above), and $x \to x^q$ is a linear function in $\mathbb{E} \to \mathbb{E}$.

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

b) Variations on HFE

The scheme described above is usually referred to as “basic HFE” or “HFE” and has been successfully broken [7]. There are some simple variations of HFE to which these attacks do not apply. These variations [3, 7] include the following.

C^{*-} is a variation on the Matsumoto-Imai-System [5] and works, in contrast to HFE, on a strictly monic polynomial for the private key. This allows faster private key operations than HFE and forms the basis of *Flash* and *SFlash* that are part of the proposed European cryptographic standard [8]. We are currently unaware of any successful attacks. The second version of HFE (HFE-), used in our implementation, removes some of the public equations and is considered to be secure. Another version, *Quartz*, included in the NESSIE proposal, is a HFEv- scheme, this means that there are more variables than necessary and it has some equations removed. It is easier to implement HFE- than HFEv-. However, further work can extend this implementation to HFEv-.

III Implementation Issues

We have implemented HFE- over $\mathbb{F} = \mathrm{GF}(4)$, $\mathbb{E} = \mathrm{GF}(64)$ and $\mathbb{F} = \mathrm{GF}(2)$, $\mathbb{E} = \mathrm{GF}(2^n)$.

There are three security parameters in HFE: first of all the degree n of the extension field, second the degree d of the private polynomial and third the cardinality q of the underlying field $\mathbb{F}$. The suggestion is to choose n to be a prime to avoid subfield attacks, d to have the form $2k + 1$, and to increase n rather than d to make the secret key operations not too difficult. To keep our operations simple we not only removed the public equations $(p_1, \ldots, p_k)$ where $1 < k \ll n$ but also added some random polynomials $(q_1, \ldots, q_k)$. This is no threat to the security of HFE [3, 7]. In our implementation we have chosen $q = 2$, $n = 67, 129$, $d = 33$ and $k = 4$ and generally follow the recommendations from [9]. The value $n = 67$ is too small for HFE but shows how our implementation scales up for bigger values of n.

The affine transformations S and T are implemented as $n \times n$ matrices and an additional n-dimensional vector over $\mathbb{F}$. To add redundancy to our message m we use randomly generated polynomials [3].

Our current implementation is a Java package that consists of approximately 14,000 lines [10]. It provides APIs for key generation, encryption, decryption, signature generation and verification. Classes and their APIs are consistent with Java’s cryptography architecture and our package has been designed for easy integration into this architecture. The flexibility of Java’s cryptography architecture means that, for example, existing SSL based applications could select HFE as the public-key scheme. Due to space limitations we describe only some aspects of our implementation.

a) Finite Field Operations

Standard algorithms [11] were used to implement the finite field operations (addition, subtraction, multiplication, division). The worst case complexity of these operations is typically polynomial in the size of the operands. In GF(64) performance was improved by storing precomputed values for division and multiplication in a lookup table. Such an approach is not feasible for $\mathrm{GF}(2^n)$ as it results in impractical table sizes. Table compression techniques such as Zech Logarithms [12] are ineffective due to the size of the extension field. For division in the extension field, we use the algorithm from [13].

b) Polynomials

HFE deals with polynomials over an extension field and with multivariate polynomials over the underlying field. For the extension field we chose a coefficient representation, that is, for every power of x we store the corresponding coefficient a in an array. Polynomial multiplication is done using a standard $O(n^2)$ algorithm; a later version might use [14, p. 798] which provides $O(n \ln^2 n)$ by using a Fast Fourier Transformation like algorithm. Our representation of the multivariate polynomials exploits the fact that they are of degree at most 2. Figure 3 illustrates how the coefficients are stored in the coefficient array.

In general, the number of terms stored in the coefficient array is $\tau(n) = ((n + 1)(n + 2))/2$ el-

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

Fig. 3: Coefficient Array of Multivariate Polynomial, $n = 3$

ements. 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. One class supports polynomials of degree 1, with operations performed in $O(n)$ space and time. Another class (using the coefficient array from Figure 3) 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.

c) Matrices

To invert S and T, we must be able to invert the underlying matrices. Parts of the public domain Jama Package [15] were rewritten to provide the necessary matrix operations over finite fields. We found that the Jama package generally provides good performance. While Jama multiplication takes $O(n^3)$ and is reasonable for conventional applications, the large matrices required by HFE are such that any improvement (such as Strassen [14, pp. 739-745]) could result in a speedup during decryption. LU decomposition is used to invert our matrices.

d) Key Size

The size of the public key is the total number of coefficients for all public polynomials. This is $n\tau(n) = n((n+1)(n+2))/2$ and leads to 157,182 and 1,098,435 coefficients for $n = 67$ and 129. These coefficients are over GF(2). If we assume that they are stored in terms of bytes, we will need at least 19kb and 134kb.

For the private key we need to store the coefficients of our private polynomial (at most 34) and the entries of two $n \times n$ matrices and two n-vectors. This leads to about 1.4kb / 4.6kb as the lower bound to store our private key. These keys are stored in a human readable, rather than size efficient, form.

IV Performance

The following timing results where obtained using a Pentium-III-730MHz running Java 1.3. They are based on 101 measurements for key generation, decryption, signature generation and 10,100 measurements for encryption and signature verification. Figure 4 gives the mean for each of these performance measurements.

Extension Field	GF(2^{67})	GF(2^{129})
Key Generation	4453 ms	63341 ms
Encryption	4.83 ms	28.98 ms
Decryption	2527 ms	12,023 ms
Signature	305 ms	1331 ms
Verification	2.47 ms	20.73 ms

Fig. 4: Running Time on a Pentium-III-730MHz

These results were obtained declaring all classes as final and with frequently used variables declared global static rather than local.

According to [16], a C++ implementation of Quartz needs 30 seconds to generate one signature on a Pentium-III-500MHz. As Quartz consists of 4 HFEv- steps over GF(2^{103}), this translates to approx. 7 seconds for one single step over GF(2^{103}). To the knowledge of the authors, this C++ Quartz implementation is not publicly available. Therefore we could not test it on the same machine as this Java HFE- implementation for speed comparison.

V Conclusions

As the latest attacks on HFE ([7, 17]) date back to 1999 and 2001, it might be too early to use HFE now. While basic HFE is considered to be broken, no successful attack exists on HFE- and HFEv-. The *Quartz* algorithm, forming part of the proposed European cryptographic standard [8], is based on HFEv-.

When compared with RSA or ElGamal, the larger storage requirements for HFE keys is a disadvantage. However, as pointed out in [3], HFE is faster than RSA. Since these results were based on specific values for the security parameters [3], the authors of this paper believe that further studies are required before HFE and RSA can be realistically compared.

On the other hand HFE does not belong to the same class of algorithms as RSA and ElGamal. Therefore attacks that are known for RSA and ElGamal do not apply to HFE. Once the polynomial equation is solved, the operations for HFE are straightforward.

Since the security parameters influence the performance of HFE, tradeoffs may be made on the relative amount of computation required for encryption and decryption, for example, for ensuring reasonable processing time when using smart-card/PDA and workstation. If the smartcard/PDA is used for encryption (and the PC decrypts) then it is better to increase d rather than n. If the smartcard/PDA is used for signature (and the PC for verification) then it is better to increase n rather than d.

References

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

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

[3] Jacques Patarin: "Hidden Field Equations (HFE) and Isomorphisms of Polynomials (IP): two new Families of Asymmetric Algorithms". *EuroCrypt '96*. Extended Version: http://www.minrank.org/hfe.pdf

[4] "Cryptographic communication process". US Patent 5,790,675, August 4, 1998.

[5] T. Matsumoto and H. Imai: "Public Quadratic Polynomial-tuples for efficient signature-verification and message-encryption". pp. 419–453, *EuroCrypt '88*, Springer Verlag 1988.

[6] Jacques Patarin: "Cryptanalysis of the Matsumoto and Imai Public Key Scheme of Eurocrypt'88". *CRYPTO'95*, pp. 248-261, 1995.

[7] Nicolas T. Courtois: "The security of Hidden Field Equations (HFE)". *Cryptographers' Track RSA Conference 2001*, http://www.minrank.org/hfesec.pdf

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

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

[10] Christopher Wolf: *Java Implementation of Hidden Field Equations.* http://www.christopher-wolf.de/hfe

[11] Julio López and Ricardo Dahab: "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

[12] Klaus Huber: "Some Comments on Zech's Logarithms". *IEEE Transactions on Information Theory*, 36(4):946–950, July 1990.

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

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest: "Introduction to Algorithms". 17^{th} printing, *The MIT Press*, McGraw-Hill Book Company, ISBN 0-262-03141-8 or 0-07-013143-0

[15] 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/

[16] J. Patarin, N. Courtois, and L. Goublin: "Submission of Quartz for NESSIE". http://www.cosic.esat.kuleuven.ac.be/nessie/workshop/submissions/quartz.zip.

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