Implementing Quartz in Java

Draft for the 3^{rd} NESSIE Workshop

Christopher Wolf
chris@Christopher-Wolf.de
Department of Mathematics, University College Cork &
Faculty of Computer Science, University of Ulm

Date: *September 5, 2002*

Abstract

Quartz is proposed for NESSIE as an asymmetric signature standard and has the advantage of a very short signature length (128 bits). This report outlines our experience implementing Quartz in Java. Moreover, it gives a speed comparison between a simplified version of Quartz and DSA (1024 bits). In addition, we further investigate public key generation and present a new algorithm that offers some improvement over the existing technique. Both have the same complexity, although our speed comparison shows that they are more efficient in terms of computational time.

1 Introduction

Quartz is a variation of Hidden Field Equations (HFE) [1], which is a family of signature and encryption algorithms. The whole family is based on the "MinRank"-problem [1], i.e., the intractability of solving quadratic polynomial equations over a finite field $\mathbb{F}$ for sufficiently many quadratic unknowns $x_1, \ldots, x_n$. In a HFE scheme, such polynomials work as a public key, while the private key consists of two affine transformations $\mathbb{F}^n \to \mathbb{F}^n$, denoted S and T, and a polynomial P over an extension field $\mathbb{E}$. It is possible to express any private key (S, P, T) in terms of at most quadratic polynomials over the underlying finite Field $\mathbb{F}$ [1, 2]. Computing $y_1, \ldots, y_n$ for given $x_1, \ldots, x_n$ is always possible using the public key. Each of the private key operations is invertible, and thus it is possible, using the trapdoor, to compute $x_1, \ldots, x_n$ for given $y_1, \ldots, y_n$. For the NESSIE project, three variations of HFE, named Flash, SFlash, and Quartz, have been proposed [3]. Modified versions of SFlash and Quartz are still under consideration [4].

In comparison to Flash, Quartz has a much shorter signature length of 128 bits. This is short for a public key signature. For example, a DSA signature with a 1024 bit modulus size is 2048 bits long [5]. Although signature generation with Quartz is rather slow, its short signature length makes Quartz an interesting public key signature algorithm [4]. The fact that Quartz is neither based on the factorization problem nor the discrete logarithm problem, increases the interest further.

This report outlines, how Quartz can be efficiently implemented in Java. As Java is available on a large number of platforms - not only workstations, but also, for example, wireless communication devices, and PDAs - it is natural to envisage that many implementations of Quartz will use Java. We are not aware of a finite field library such as NTL [6] or LiDIA [7] for Java. Thus the implementation of Quartz had to be done "from scratch", based on the original specification from [8, 1, 9]. For this reason, this report focuses on a simplified version of Quartz, called QuartzLight (see Section 2). Our performance results for this simpler implementation can be expected to be comparable with the performance of Quartz in general. However, QuartzLight is expected to be less secure than Quartz and therefore not recommended for signature.

This report is organized as follows: in Section 2, we give a short description of Quartz and QuartzLight. In Section 3, we describe the Java implementation of QuartzLight, stressing how good performance can be achieved in a Java environment. We also point out, how Quartz itself can be implemented. Section 4 concentrates on the issue of generating the public key for a given private key and describes a new algorithm for key generation. The overall performance of this implementation is discussed in Section 5. This paper concludes with Section 6. The appendix shows how to compute the public key for a given private key, using the algorithm from Section 4.2.

2 Quartz and QuartzLight

Quartz consists of 4 rounds of a HFEv- scheme, i.e., a Hidden Field Equation problem with vinegar variables and some publicly known equations removed. Let denote n the dimension of the extension field $\mathbb{E} = \mathbb{F}[t]/i(t)$, where $i(t)$ is an irreducible polynomial from $\mathbb{F}[t]$, and the degree of $i(t)$ is n. The number of vinegar variables is denoted by v. The number of elements in $\mathbb{F}$ is denoted by q. With this notation, the affine transformation S is over $\mathbb{F}^{n+v} \to \mathbb{F}^{n+v}$, while the affine transformation T is over $\mathbb{F}^n \to \mathbb{F}^n$. The private polynomial P in a HFE vinegar scheme is a family of functions in the vinegar variables $(z_1, \ldots, z_v) \in \mathbb{F}^v$. The degree of these polynomials is d. Polynomial P is defined as:

$$\begin{aligned} P_{(z_1,\ldots,z_v)} \quad := \quad & \sum_{\substack{0 \le i,j \le d \\ q^i+q^j \le d}} \alpha_{i,j} x^{q^i+q^j} + \sum_{\substack{0 \le k \le d \\ q^k \le d}} \beta_k(z_1, \ldots, z_v) x^{q^k} + \gamma(z_1, \ldots, z_v) \\ & \text{for } \alpha_{i,j} \in \mathbb{E}, \\ & \beta_k(z_1, \ldots, z_v) \text{ are affine in } (z_1, \ldots, z_v), \text{ and} \\ & \gamma(z_1, \ldots, z_v) \text{ is at most quadratic in } (z_1, \ldots, z_v) \end{aligned}$$

The function $\mathrm{HFE}(x) := T(P(S(x)))$ can be expressed in terms of at most quadratic polynomials $(p_1, \ldots, p_n)$ over $\mathbb{F}$ [1]. Each of these polynomials has $(n+v)$ input variables $(x_1, \ldots, x_{n+v})$ over $\mathbb{F}$. For HFE-, some of these n polynomials are removed. The conrete parameters of Quartz are [8]:

1. Finite Field: $\mathbb{F} = \mathrm{GF}(2)$
2. Vinegar variables: 4
3. Extension Field: $\mathbb{E} = \mathrm{GF}(2^{103})$
4. Polynomials removed: 3

In addition, Quartz also uses a so called "Feistel-Patarin-network" [8]. This means that the HFEv- part of Quartz is used four times rather than only once. So to obtain a signature in Quartz, 4 roots of the private polynomial $P \in \mathbb{E}[t]$ have to be computed. To verify a signature, it is necessary to evaluate the public key 4 times and to compare the result with the given signature. A signature is only accepted, if all 4 evaluations of the public key yield the same result as in the signature. The "Feistel-Patarin-network" is also designed in a way, that only 128 bit rather than 400 bits are needed for a signature. See [8] for further details.

In contrast, QuartzLight (see Figure 1) is a HFE- scheme, i.e., Hidden Field Equations with some public equations removed but without vinegar variables. As [1, 10] point out, this modification is expected to be less secure than HFEv- but of similar speed. We did not implement four invocations of the underlying HFE- problem, but only 1, and the structure of the "Feistel-Patarin-network" is omitted. This is not expected to be a problem in terms of

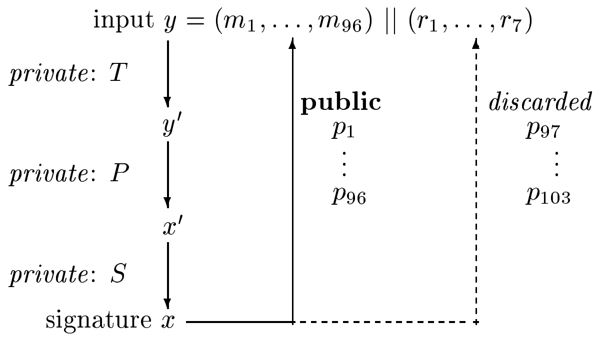

Fig. 1: Overview of QuartzLight

speed comparison, as we expect a "Feistel-Patarin-network" to have four times the running time of a single HFE- step. In Figure 1, $\cdot||\cdot$ denotes the concatenation function, $m_1, \ldots, m_{96}$ are the input message, and $r_1, \ldots, r_7$ are "randomly" set, free bits. These free bits are necessary to obtain a valid signature with a reasonably high probability, as HFE(x) is not a surjection [1, 8].

3 Implementation Issues

For our implementation, we had two conflicting goals. On one hand, the implementation is meant to be as close to Quartz as possible, so that the speed measurements can be compared with Quartz. On the other hand, Quartz consists of many different operations that do not impact the performance. As a compromise, we concentrated on the features of Quartz, which are computationally expensive. In particular, the degree of the extension field $\mathbb{E}$ ($n = 103$) and the degree of the private polynomial P ($d = 129$) are the same. In contrast, QuartzLight is a HFE- scheme rather than a HFEv- scheme. This means, we did not implement vinegar variables. We also did not implement the Feistel-Patarin-network, as this is a rather fast security feature and we do not expect it to have a great impact on the speed of our implementation. We are confident that our implementation provides a good estimation of the speed of Quartz in a Java environment.

3.1 Finite Field Arithmetic

QuartzLight uses two different finite fields, namely $\mathbb{F}$=GF(2) and $\mathbb{E}$=GF(2^{103}). Both were implemented using standard techniques from [11, 12]. For $\mathbb{E}$, speed up techniques such as Zech Logarithms [13] were considered but rejected due to the impractical table sizes ($\approx 10^{33}$ or $\approx 2^{110}$ bits).

3.2 Polynomials

QuartzLight requires a polynomial of degree 129 over GF(2^{103}), and also polynomials in 103 variables over GF(2). Figure 2 describes our implementation for multivariates polynomials (that is, with mixted terms of the form $x_i x_j$) which are used for representing the public key. As these polynomials are over GF(2), $x_i x_i = x_i$, when $i \neq j$. For performance reasons, multivariate polynomials were implemented in two different classes, one for strictly linear polynomials ($1 + 103 = 104$ coefficients) and one for quadratic polynomials ($1 + 103 + \frac{103 \cdot 102}{2} = 5357$ coefficients), so arithmetic operations involving two linear polynomials require far less time than the same operations with two quadratic polynomials. We found the most expensive operation to

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

Fig. 2: Coefficient Array of Quadratic Polynomial, for 3 variables

be multiplication of two affine polynomials. Using an interpolation like approach saved about half the finite field multiplications required and, therefore, gave much better performance. For fast evaluation of quadratic polynomials, we used a vector $(x_1, \ldots, x_n, x_1x_2, \ldots, x_{n-1}x_n)$, which was computed once and then applied to all polynomials in the public key. For private polynomial P we also wrote a hand optimized evaluation function as this operation is frequently used during key generation (see Section 4.1).

3.3 Matrices

To deal with the matrices in the two affine transformations S and T, we rewrote parts of the public domain Java Matrix package JAMA [14] to handle finite field arithmetic. We found that JAMA gives good performance in general. However, for the large matrices involved, it would benefit from fast matrix multiplication algorithms (such as Strassen [15, pp. 739-745]), especially during key generation.

3.4 Root Finding

As root finding is the most time consuming task in QuartzLight (and also Quartz), we devoted much time optimizing this step. We found that the approach from [8] gives reasonable performance when implemented with fast algorithms. The key idea is first to compute $g(x) := \gcd(x^{|\mathbb{E}|} - x, P - y)$ for given y, and then to find all roots of $g(x)$ with an arbitrary root finding algorithm. For a proof that this technique works, and also the root finding algorithm, see [16]. In essence, we implemented a register machine with 12 registers, where each register holds one polynomial from $\mathbb{E}[t]$. This structure avoids expensive memory allocation and also copy operations. In addition, the frequent operation of squaring polynomials modulo P was done using algorithms from [11].

3.5 Conclusions

Using standard techniques, it was relatively straightforward to implement signature generation and signature verification for QuartzLight. We expect Quartz to be suitable for a Java environment. Generally speaking, it was vital to avoid dynamic memory allocation during the algorithm, as we found this being rather time consuming in Java. This led to many variables being declared static global rather than local in our implementation.

To implement Quartz, both the "Feistel-Patarin-network" and the 'v' modification in HFE must be taken into consideration. The Feistel-Patarin-network is relatively straightforward to implement, as it mainly consists of 4 invocation of the underlying HFEv- scheme. In addition, it uses the SHA-1 hash algorithm to obtain cryptographically secure pseudorandom numbers in various parts of Quartz.

For Quartz, we also need to implement the 'v' modification. As pointed out in [8], it is possible to deal with this problem by storing $q^v = 2^4 = 16$ different polynomials, where the coefficients $\beta_k(z_1, \ldots, z_v) \in \mathbb{E}$ depend on each other in an affine way, while $\gamma(z_1, \ldots, z_v) \in \mathbb{E}$ also has quadratic dependence.

4 Public Key Generation

After we discussed general implementation issues in the last section, we will now concentrate on the important step of public key generation. The proposed standard [8] does not give many details how to obtain the public key from a given private key. However, [2] provides a detailed description for the Matsumoto-Imai scheme, which has the same kind of public key as the HFE family.

4.1 Polynomial Interpolation

The approach discussed in [2] to generate the public keys for a given private key can be summarized as polynomial interpolation for multivariate polynomials. By choosing all vectors from $\mathbb{F}^{103}$ with Hamming weight less than or equal to 2, the computation of the public key becomes very simple: evaluating the vector with Hamming weight 0 gives the constant terms for all public key polynomials, evaluating the vectors with Hamming weight 1 yields their linear terms, and finally evaluating the vectors with Hamming weight 2 leads to the quadratic terms.

For this interpolation, it is necessary to evaluate HFE(x) $:= T(P(S(x)))$ in $1 + n + \frac{n(n-1)}{2}$ vectors where n denotes the degree of the extension field $\mathbb{E}$. Each affine transformation S and T requires $O(n^2)$ steps, which yields $O(n^4)$ operations for the first step, namely, evaluating S. The second step, that is, evaluating P, requires each time $O(n^2)$ multiplications - due to the special choice of the polynomial P. In addition, $O(\log d)$ squaring operations are necessary. As each multiplication requires $O(n^2)$ operations [11], and we also evaluate $O(n^2)$ values each, then the overall complexity of key generation with polynomial interpolation is $O(n^6)$, as $O(\log d)$ is negligible compared with $O(n^2)$.

4.2 Base Transformation

As pointed out in the previous section, it requires $O(n^6)$ steps to obtain the public key from a given private key by using polynomial interpolation. In this section, we describe a new algorithm for this problem, called "base transformation". The key idea (see the appendix for a toy example) is to transfer a base of the message space $\mathbb{F}^n$ rather than $O(n^2)$ vectors. Using base transformation, we obtain the same complexity as for polynomial interpolation. However, the first step, that is, applying affine transformation S, will require only $O(n^2)$ steps.

As already pointed out, we do not apply HFE(x) to elements from $\mathbb{F}^n$, but to an arbitrary base $\mathfrak{B}$ of $\mathbb{F}^n$. In polynomial notation, the base chosen is

$$\mathfrak{B} = \left\{ \begin{array}{ccccc} p_1 & = & x_1, & & \\ & \vdots & & \ddots & \\ p_n & = & & & x_n \end{array} \right\}$$

and consists of n polynomials. Furthermore, base transformation identifies this base with an element of $\mathbb{E} = \mathbb{F}[t]/i(t)$. Here $i(t)$ means the irreducible polynomial which generates $\mathbb{E}$. So the set $\mathfrak{B}$ is also viewed as polynomial

$$\mathfrak{P}(x_1, \ldots, x_n)[t] = p_n t^{n-1} + \ldots + p_1 = x_n t^{n-1} + \ldots + x_1$$

Another way of thinking about this polynomial $\mathfrak{P}$ is to replace each coefficient for an element a_i for given $a \in \mathbb{E}$ by the corresponding polynomial p_i. In this context, it is important to notice the difference between t on the one hand and $x_1, \ldots, x_n$ on the other hand. The first generates the finite field $\mathbb{E}$, while the second is a base of the message space $\mathbb{F}^n$. All operations in HFE$(x) = T(P(S(x)))$ are applied in the same way as they would be applied to elements from $\mathbb{E}$. This means especially that multiplication and squaring are done modulo $i(t)$. The following sections goes into more details and also deals with the complexity of the operations involved.

4.2.1 Affine Transformation S

When applying S to $\mathfrak{P}$, each coefficient in $\mathfrak{P}$ is multiplied with n elements from the corresponding matrix, so we would expect a total of $O(n^3)$ operations. By virtue of the choice of $\mathfrak{P}$, this drops to $O(n^2)$, as each polynomial $p_1, \ldots, p_n$ has only one non-zero coefficient. Exploiting the choice of $\mathfrak{P}$ further, transferring S to arbitrary polynomials $p_1, \ldots, p_n$ can be seen as a simple copy operation and hence requires $n^2 + n = O(n^2)$ operations in total.

4.2.2 Applying Polynomial P

After expressing affine transformation S in terms of affine polynomials $p_1, \ldots, p_n$, that is, in terms of $\mathfrak{P}$, we have to investigate, how raising $\mathfrak{P}$ to the power of $q := |\mathbb{F}|$ affects the underlying polynomials. For this purpose, we concentrate on a polynomial $p = a_0 + a_1x_1 + \ldots a_nx_n$ with $a_i \in \mathbb{F}$. Using $x^q = x$ and $(a+b)^q = a^q + b^q$ (see [16]), we obtain

$$\begin{aligned} p^q &= (a_0 + a_1x_1 + \ldots a_nx_n)^q \\ &= a_0^q + a_1^qx_1^q + \ldots a_n^qx_n^q \\ &= a_0 + a_1x_1 + \ldots a_nx_n \end{aligned}$$

so $p^q = p$ for all polynomials over $\mathbb{F}$. However, $\mathfrak{P}$ also depends on t, and $t^q \neq t$ in general. So $\mathfrak{P}^q$ will yield an arbitrary $\mathfrak{Q}$, which consisits of only linear polynomials in $x_1, \ldots, x_n$. Computing $x^{q^i + q^j}$ however, is by no means a linear operation but yields quadratic polynomials in $x_1, \ldots, x_n$. Last, we have to apply the coefficients, that is, elements from $\mathbb{E}$, to the result. This requires a further reduction by t but does not change the degree of the polynomials involved.

In terms of complexity, squaring of a polynomial $\mathfrak{P}(x_1, \ldots, x_n)[t]$ requires $O(n^3)$ steps. Multiplying two affine polynomials $\mathfrak{P}$, $\mathfrak{Q}$ is more expensive: it requires $O(n^2)$ multiplications over $\mathbb{E}$ - both for interpolating the result $\mathfrak{R}$ or for computing it directly - and each multiplications requires $O(n^2)$ steps in terms of $\mathbb{F}$. As private polynomial P has $O(n^2)$ many coefficients, the overall complexity is $O(n^6)$. Adding the results $\mathfrak{R}$ from these computations will yield only a complexity of $O(n^4)$ and is hence negligible.

4.2.3 Transformation T

As for affine transformation S, we have to apply affine transformation T to the result of Section 4.2.2. This requires $O(n^4)$ operations, as we have to multiply $O(n^2)$ many coefficients with an $(n \times n)$-matrix.

4.2.4 Overall Algorithm

Using the different steps outlined above, we obtain the following algorithm:

1. Express S in terms of affine polynomials, that is, as $\mathfrak{P}$.
2. Compute $\mathfrak{P}^{q^i}$ for $i = 0, 1, \ldots \lfloor \log_2 d \rfloor$
3. Compute $\mathfrak{P}^{q^k + q^l} = \mathfrak{P}^{q^k}\mathfrak{P}^{q^l}$ using the results of step 2.

4. Compute $\beta_i \mathfrak{P}^{q^i}$ and $\alpha_{k,l} \mathfrak{P}^{q^k+q^l}$ 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.

Here $\beta_i, \alpha_{j,k} \in \mathbb{E}$ are the coefficients of the private polynomial P. As QuartzLight is a HFE- scheme, they do not depend on any variables but are constant.

4.3 Adapted Evaluation as Intermediate Technique

As we saw in sections 4.1 and 4.2, it is possible to compute the public key for a given private key in $O(n^6)$ steps. The advantage of the base transformation technique is that the number of steps is reduced from $O(n^4)$ to $O(n^2)$ when applying the affine transformation S.

However, it is also possible to obtain a similar result without using the base transformation technique. For this we observe that any vector of Hamming weight two is the sum of two vectors of Hamming weight one. Denote $\eta_i \in \mathbb{F}^n$ the vector which has only zeros but one at position i and $\eta_{i,j} \in \mathbb{F}^n$ the vector which has only zeros but one at positions i, j where $1 \le i \le n$ and $1 \le i < j \le n$, respectively. Moreover, denote η_0 the vector which has zeros only. Using this, we define further:

$$s_0 := S(\eta_0) \quad s_i := S(\eta_i) \quad s_{i,j} := S(\eta_{i,j})$$

It is easy to see that $s_{i,j} = s_i + s_j - s_0$. In addition, applying S to s_0, s_i does not require any matrix multiplication as s_i can be seen as selecting one column vector rather than a complete matrix multiplication. So obtaining s_0, s_i requires $O(n^2)$ steps, while computing $s_{i,j}$ requires $O(n^2)$ additions, and hence has complexity $O(n^3)$, as each addition requires $O(n)$ steps [11].

In addition, we can exploit the fact that $(a+b)^{q^k} = a^{q^k} + b^{q^k}$ in finite fields (see [16]). So it is sufficient to raise s_0, s_i to the power $q^1, \ldots, q^{\lfloor \log d \rfloor}$ and then obtain $s_{i,j}^{q^k}$ by adding the corresponding elements. Again, we obtain complexity $O(n^3)$, as we have to perform $O(n^2)$ additions.

In terms of complexity, this is worse than base transformation. However, for base transformation, we need much more specialized code, while this adapted evaluation technique only requires minor additional code. So in terms of the implementation effort versus performance trade-off, adapted evaluation seems to be the best choice of the three options discussed in this section.

4.4 HFEv

As QuartzLight is a HFE- scheme, we concentrated on key generation for this type of HFE scheme. However, both base transformation and adapted evaluation can be used in a HFEv scheme. For base transformation, we need to distinguish between replacing terms and multiplying terms as this obviously yields different results. To adapt the base to HFEv, we think of the 4 vinegar variables in Quartz as four polynomials, namely $p_{104}, \ldots, p_{107}$, which depend on 107 variables each. For the "constant" term, we will get 103 quadratic polynomials in 107 variables each, by replacing the vinegar variables $z_i z_j$ with the corresponding polynomials. Consistently, we gain affine polynomials for the linear terms in P. The other operations are modified in a similar way.

For adapted evaluation, we have to treat the four most significant bits from each vector $s_0, s_1, \ldots, s_n, s_{1,2}, \ldots, s_{n-1,n}$ as vinegar variables. Using 16 different, precomputed private polynomials $P_0, \ldots, P_{15}$ - depending on the 4 vinegar variables $z_1, \ldots, z_{,4}$, can speed up the evaluation of HFE(x) further.

5 Performance

All speed measurements for the given code [17] were done on a Pentium III-930MHz with Java 1.3.

5.1 Key Generation

Algorithm	Mean (s)	Range (s)
DSA (1024 bits / SHA-1)	31.5	0.89 - 162
QuartzLight Base Transformation	2.6	2.5 - 2.9
QuartzLight Polynomial Interpolation	3.2	3.1 - 3.9

The table above is based on 101 measurements for each key generation algorithm. Mean is the average of these measurements, while range gives both the least and the hightest value for each individual measurement. To have a fair comparision, we did not use the precomputed values for DSA key generation.

5.2 Signature

	DSA (1024 bits / SHA-1)		QuartzLight	
	Mean (ms)	Range (ms)	Mean (ms)	Range (ms)
Signature Generation	9.5	9.4 - 10.1	5700	4000 - 16,000
Signature Verification	18.9	18.2 - 19.4	9.9	9.9 - 9.9

The table above is based on 10,100 measurements per algorithm. Due to the limited accuracy of the timer used, 100 measurements were grouped together.

5.3 Discussion

Key generation for Quartz is clearly much faster than for DSA, regardless of whether the keys for Quartz are computed using base transformation or polynomial interpolation. In addition , the running time for base transformation is ≈20% lower than that for polynomial interpolation. As both algorithms have complexity $O(n^6)$ in total, we would have expected a far more similar running time for both algorithms. We assume that the reason for this result lies in the fact, that the code for base transformation can be optimized further than the code for polynomial interpolation. While the latter consists of very simple, easy to optimize steps, the first consists of many different steps, which can be optimized using various techniques, for example, using an interpolation-like technique for multiplication of affine polynomials.

In terms of signature generation and verification, QuartzLight is certainly quite slow in terms of genration, but quicker than DSA in terms of signature verification. Using the figures from above, we estimate Quartz to have a key generation time of ≈3 s (similar to QuartzLight), a signature verification time of ≈40 ms (four times QuartzLight), and a signature generation time of ≈23 s (four times QuartzLight). According to our measurements, one invocation of SHA-1 takes less than $10\mu s$ in Java. Hence we do not expect the structure of a Feistel-Patarin-network to have impact on this result.

6 Conclusions

As we saw in this paper, generating the public key of Quartz with our new technique "base transformation" has advantages in terms of speed, compared with the traditional polynomial interpolation, it is approximatively 20% faster (see Figure 3).

	Polynomial Interpolation	Base Transformation
Affine Transformation S	$O(n^4)$	$O(n^2)$
Private Polynomial P	$O(n^6)$	$O(n^6)$
Affine Transformation T	$O(n^4)$	$O(n^4)$
Overall Complexity	$O(n^6)$	$O(n^6)$
Running Time (mean)	3.2 s	2.6 s

Fig. 3: Summary of Results for Key Generation Algorithms

In addition, key generation for Quartz is expected to be 12 times faster than for DSA, which needs approximatively 31.5 s on average for generating one public / private key pair (see Section 5).

The picture changes if we compare these two algorithms in terms of signature generation and signature verification (see Figure 4). Here, DSA is certainly faster than Quartz. However, as [4] points out, Quartz has the advantage of having a very short signature length, namely 128 bits.

	Signature Generation	Signature Verification
DSA (1024 bits / SHA-1)	9.5 ms	18.9 ms
Quartz (Estimation)	23,000 ms	40 ms

Fig. 4: Summary of Results for Signature Algorithms

As an overall result, Quartz is certainly suitable for a Java environment. The high signature generation time could be a problem for certain applications domains, for example, signing documents on a PDA, as these devices have only a limited computational power. In other application domains, for example, having a mobile phone verifying a signature computed by a server, the very low signature length will be a great advantage. When an application requires a high number of public / private key pairs to be computed, generating this pair with base transformation rather than polynomial interpolation is an attractive option as the computational time is far lower. Due to the fact that Quartz and SFlash have a very similar public key structure [3], base transformation is also a worthwhile option for public key generation in SFlash.

7 Acknowledgements

I want to thank Simon Foley (Dept. of Computer Science, University College Cork, Ireland) for suggesting to write this article and his comments. In addition, I want to thank Patrick Fitzpatrick (Dept. of Mathematics, University College Cork, Ireland) for his comments.

Appendix

Example of Public Key Generation

This section gives a toy example of public key generation, using the algorithm from Section 4.2. The paramters are: $\mathbb{F}$ =GF(2), $i(t) := t^3 + t + 1$, $\mathbb{E} = \mathbb{F}[t]/i(t)$, and $P(x) = x^3 + (t+1)x + 1$. Here $(t+1) \in \mathbb{E}$; as as bit string, it would be denoted $[011]_b$. Moreover, let

$$\begin{aligned}
S(x_1, x_2, x_3) &:= \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \\
T(x_1, x_2, x_3) &:= \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}
\end{aligned}$$

By applying the algorithm from Section 4.2, we obtain

$$\mathfrak{P}(x_1, x_2, x_3) = \begin{pmatrix} p_1 & := & x_1 + x_3 \\ p_2 & := & x_2 + 1 \\ p_3 & := & x_2 + x_3 \end{pmatrix}$$

as a representation of S in terms of three polynomials p_1, p_2, p_3. This can equally be seen as

$$\mathfrak{P}[t] = (x_2 + x_3)t^2 + (x_2 + 1)t + (x_1 + x_3)$$

to stress the point that we want to mimic operations in $\mathbb{E}$. First we computate $\mathfrak{P}[t]^2$:

$$\begin{aligned}
\mathfrak{P}[t]^2 &= [(x_2 + x_3)t^2 + (x_2 + 1)t + (x_1 + x_3)]^2 \\
&= (x_2 + x_3)t^4 + (x_2 + 1)t^2 + (x_1 + x_3) \\
&\equiv (x_3 + 1)t^2 + (x_2 + x_3)t + (x_1 + x_3)
\end{aligned}$$

where $\equiv$ denotes reduction by $i(t) := t^3 + t + 1$. Furthermore,

$$\begin{aligned}
\mathfrak{P}[t]^3 &= \mathfrak{P}[t]^2 \mathfrak{P}[t] \\
&= [(x_3 + 1)t^2 + (x_2 + x_3)t + (x_1 + x_3)][(x_2 + x_3)t^2 + (x_2 + 1)t + (x_1 + x_3)] \\
&= (x_2 + x_2x_3)t^4 + (x_2x_3 + 1)t^3 + (x_1 + x_1x_2)t^2 + (x_1 + x_1x_3)t + (x_1 + x_3) \\
&\equiv (x_1 + x_2 + x_1x_2 + x_2x_3)t^2 + (x_1 + x_2 + x_1x_3 + 1)t + (x_1 + x_3 + x_2x_3 + 1)
\end{aligned}$$

As x in $P(x)$ has a constant multiple $(t+1)$, we have to reflect this in terms of $\mathfrak{P}[t]$:

$$\begin{aligned}
(t+1)\mathfrak{P}[t] &= (t+1)[(x_2 + x_3)t^2 + (x_2 + 1)t + (x_1 + x_3)] \\
&= (x_2 + x_3)t^3 + (x_3 + 1)t^2 + (x_1 + x_2 + x_3 + 1)t + (x_1 + x_3) \\
&\equiv (x_3 + 1)t^2 + (x_1 + 1)t + (x_1 + x_2)
\end{aligned}$$

So as an overall result, we can compute $P(x)$ in terms of $\mathfrak{P}[t]$ and denote the result with $\mathfrak{Q}[t]$:

$$\begin{aligned}
\mathfrak{Q}[t] &:= \mathfrak{P}[t]^3 + (t+1)\mathfrak{P}[t] + 1 \\
&= [(x_1 + x_2 + x_1x_2 + x_2x_3)t^2 + (x_1 + x_2 + x_1x_3 + 1)t + (x_1 + x_3 + x_2x_3 + 1)] + \\
&\quad [(x_3 + 1)t^2 + (x_1 + 1)t + (x_1 + x_2)] + 1 \\
&= (x_1 + x_2 + x_3 + x_1x_2 + x_2x_3 + 1)t^2 + (x_2 + x_1x_3)t + (x_2 + x_3 + x_2x_3)
\end{aligned}$$

Before we can apply affine transformation T, we have to change our point of view and to think about $\mathfrak{Q}[t]$ as a vector with 3 rows, denoted $\mathfrak{Q}(x_1, x_2, x_3)$:

$$\mathfrak{Q}(x_1, x_2, x_3) := \left(\begin{array}{lll} q_1 & := & x_2 + x_3 + x_2 x_3 \\ q_2 & := & x_2 + x_1 x_3 \\ q_3 & := & x_1 + x_2 + x_3 + x_1 x_2 + x_2 x_3 + 1 \end{array} \right)$$

Affine transformation T can now be applied by ordinary matrix multiplication and vector addition. This step yields result $\mathfrak{R}$:

$$\begin{aligned} \mathfrak{R}(x_1, x_2, x_3) &:= T(\mathfrak{Q}(x_1, x_2, x_3)) \\ &= \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \\ &= \left(\begin{array}{l} q_2 + q_3 + 1 \\ q_1 + q_2 \\ q_3 \end{array} \right) \\ &= \left(\begin{array}{l} x_1 + x_3 + x_1 x_2 + x_1 x_3 + x_2 x_3 \\ x_3 + x_1 x_3 + x_2 x_3 \\ x_1 + x_2 + x_3 + x_1 x_2 + x_2 x_3 + 1 \end{array} \right) \end{aligned}$$

As we see, each row in the vector $\mathfrak{R}$ consists of one polynomial with at most quadratic terms in x_1, x_2, x_3. By construction, $\mathfrak{R}(x_1, x_2, x_3) = T(P(S(x_1, x_2, x_3)))$ for any $(x_1, x_2, x_3) \in \mathbb{F}^3$.

References

[1] Patarin, Jacques: *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`

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

[3] 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`,
`...workshop/submissions/flash.zip`,
`...workshop/submissions/sflash.zip`

[4] B. Preneel, A. Bosselaers, S.B. Örs, et. al: *Update on the selection of algorithms for further investigation during the second round*, Document NES/DOC/ENS/WP5/D18/1, March 2002, `http://www.cosic.esat.kuleuven.ac.be/nessie/deliverables/D18_1.pdf`

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

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

[7] TU Darmstadt, LiDIA Group: *LiDIA: A C++ Library For Computational Number Theory* `http://www.informatik.tu-darmstadt.de/TI/LiDIA/`

[8] Jaques Patarin, Nicolas Courtois and Louis Goublin: *Update of Quartz* `www.cosic.esat.kuleuven.ac.be/nessie/updatedPhase2Specs/quartz/quartzv21-b.zip`

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

[10] 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`

[11] 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`

[12] 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`

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

[14] 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/`

[15] Thomas Cornmen, 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

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

[17] Christopher Wolf: *Java Implementation of QuartzLight Source Code*. `http://www.christopher-wolf.de/ql`