Efficient Implementation of the McEliece Cryptosystem

Takuya Sumi† Kirill Morozov‡ Tsuyoshi Takagi‡

†Department of Mathematics, Kyushu University, 744, Motooka, Nishi-ku, Fukuoka, 819-0395, Japan

‡Institute of Mathematics for Industry, Kyushu University, 744, Motooka, Nishi-ku, Fukuoka, 819-0395, Japan

Abstract The code-based McEliece public key cryptosystem (PKC) is a prospective candidate for postquantum encryption. This paper presents a C++ implementation of the McEliece PKC for security parameters ($m = 11, n = 2048, t = 31, 51, 81$) and ($m = 12, n = 4096, t = 41, 62, 101$), for $k = n - mt$ in each parameter set. For example, the parameter set ($n = 2048, t = 51$) provides the security level equivalent to 95-bit AES, while the others are chosen for comparision in various ways. Key generation and decryption are not much optimized, moreover, we only implement the decoding of the Goppa code, and we only report the timing of decoding. For example, in our implementation with ($n = 2048, t = 51$), key generation takes about 47 seconds, encryption about 1 millisecond, and decryption (decoding) about 278 milliseconds. For comparison to the previous works, for ($n = 2048, t = 50$), Strenzke at WISTP 2010 reports an implementation of a variant of the McEliece PKC (which is somewhat more complicated than the original one), where encryption is performed in 1.2 millisecond on an environment comparable to ours with 2GHz CPU against 3.2 GHz CPU with us. However his decryption timing is 1.6 milliseconds, which rules our implementation not competitive in this aspect. Our future work is to complete and to optimize our current implementation.

1 Introduction

The McEliece PKC [20] is a prospective candidate for the future public key encryption [3, App. A]. Shor's algorithm can be implemented on a quantum computer to break many PKC's: RSA, ElGamal and their derivatives, which appear in standards, and which are widely used in practice. Quantum computers currently exist only as "proof-of-concept" prototypes, but full-scale devices may appear in the upcoming decades.

The syndrome decoding problem [1] cannot be efficiently solved even by quantum computers. Security of the McEliece PKC is based on hardness of the problems related to syndrome decoding [26]. In details, the syndrome decoding problem has long known to be NP-complete [1], however the security of this PKC is based on hardness of bounded distance decoding of a permuted Goppa code [11, 19, 23]. Breaking of the McEliece PKC is believed to be infeasible for properly chosen parameters [9, 10, 3].

This paper presents a C++ implementation of the McEliece PKC for security parameters ($m = 11, n = 2048, t = 31, 51, 81$) and ($m = 12, n = 4096, t = 41, 62, 101$), for $k = n - mt$ in each parameter set. Key generation and decryption are not much optimized, moreover, we only implement the decoding of the irreducible Goppa code, and we only report the timing of decoding.

In the encryption algorithm, we use the SSE2 and SSE4a operations [22] for optimizing matrix multiplication. For decryption, we use Patterson's decoding algorithm [25] for binary irreducible Goppa codes. In fact, we use standard efficient algorithms for finite fields: the Extended Euclidean algorithm for computing inverses, a square root algorithm [13, p. 136] (see also [4]) with table lookup and the Cantor-Zassenhaus algorithm for finding roots of a polynomial [6, p. 127].

For example, the parameter set ($n = 2048, t = 31$), provides the security level equivalent to 80-bit AES, a commonly accepted minimum level of security. Key generation, encryption and decoding take about 30 seconds, 1 millisecond and 116 milliseconds, respectively. At the same time, the parameter set ($n = 2048, t = 51$) provides 95-bit security, but key generation now takes about 47 seconds and decoding about 278 milliseconds. This is about 1.57 and 2.4 times more, respectively, as compared to the previous parameter set, while encryption takes about the same time.

In the recent years, a few implementations of the McEliece PKC and its variants have been reported. The works by Döring [7], Biswas [4], Strenzke [27], and Hoffmann [15] use a personal computer with CPU. An implementation on GPU is introduced by Howenga [16]. Implementations for embedded and memory-constrained devides are presented by Eisenbath et al [8], Strenzke [27], and Heyse [12]. Side-channel and power analysis attacks are studied by Heyse et al [14] (see also the references therein).

In general, it is hard to compare the CPU implementations to ours, as well as between each other, because they implement different modifications of the McEliece PKC. For a rough comparison, we chose the CPU implementation by Strenzke [27, Sec. 2], which implements the modification of the McEliece PKC by Overbeck [24]. In fact, this work provides timing for both smartcards and the personal computer with CPU – we are only interested in the latter. Note that this implementation is more involved as compared to ours. For ($n = 2048, t = 50$), Strenzke reports [27, Tab. 2] encryption in 1.2 millisecond on an environment comparable to ours with 2GHz CPU against 3.2 GHz CPU with us. However, his decryption timing is 1.6 milliseconds, which rules our implementation not competitive in this aspect. Some more details are provided in Section 4.1.

Our future work is to complete and to optimize our current implementation.

2 McEliece Cryptosystem

2.1 Notation and Definitions

For some $q, k, n \in \mathbb{N}$, when working over $\mathbb{F}_q$, a set of vectors of size n is denoted by $\mathbb{F}_q^n$, and a set of $k \times n$ matrices is denoted by $\mathbb{F}_q^{k\times n}$. We write $\mathbf{x} \in \mathbb{F}_q^n$ as $(x_1, \ldots, x_n)$.

For an ordered subset $\{j_i, \ldots, j_m\} = J \subseteq \{1, \ldots, n\}$ we denote the vector $(x_{j_1}, \ldots, x_{j_m}) \in \mathbb{F}_q^m$ by $\mathbf{x}_J$. Similarly, we denote by M_J the submatrix of a $(k \times n)$ matrix M consisting of the columns corresponding to the indices of J.

The identity matrix of size n is written as I_n. Denote by $Diag_n(\cdot)$ the diagonal matrix of size n with the arguments being the elements of the main diagonal. Denote by "$[X|Y]$" a concatenation of matrices X and Y of appropriate size.

Denote by "$\oplus$" the bitwise exclusive-or.

A q-ary (n, k)-code $\mathcal{C}$ over a finite filed $\mathbb{F}_q$ is a k-dimensional subspace of the vector space $\mathbb{F}_q^n$; n and k are called the *length* and the *dimension* of the code, respectively. We call $\mathcal{C}$ an (n, k, d)-code, if its so-called *minimum distance* is $d := \min_{\mathbf{x},\mathbf{y}\in\mathcal{C}} d_H(\mathbf{x}, \mathbf{y})$, where d_H denotes the Hamming distance (i.e. the number of position where $\mathbf{x}$ and $\mathbf{y}$ are different). The distance of $\mathbf{x} \in \mathbb{F}_q^n$ to the zero-vector $\mathrm{wt}(\mathbf{x}) := d_H(\mathbf{0}, \mathbf{x})$ is called the *weight* of $\mathbf{x}$.

We will work with the tower of finite fields $\mathbb{F}_{2^{mt}} \simeq \mathbb{F}_{2^m}[X]/g(X)$, where $g(X) \in \mathbb{F}_{2^m}[X]$ is irreducible and $\deg(g(X)) = t$ for some $m, t \in \mathbb{N}$ defined in the next section.

2.2 Key Generation

In the McEliece PKC, the secret key is an (n, k, d) irreducible binary Goppa code [11, 19, 23] correcting up to t errors. The security parameters are (n, t). Such the code is defined by the *code support* $\mathbf{L} = (\gamma_0, \ldots, \gamma_{n-1}) \in \mathbb{F}_{2^m}^n$, where $m = \log_2 n$, and an irreducible polynomial $g(X) \in \mathbb{F}_{2^m}[X]$ of degree t. We set $k = n - mt$.

Key generation is described in Algorithm 1.

Algorithm 1 Key Generation

```
INPUT: $\mathbf{L} = (\gamma_0, \ldots, \gamma_{n-1}) \in \mathbb{F}_{2^m}^n$, $g(X) \in \mathbb{F}_{2^m}[X]$ with $\deg(g(X)) = t$
OUTPUT: The public key $pk = (G^{pub} \in \mathbb{F}_2^{k\times n}, t)$; the secret key $sk = (\mathbf{L}, g(X), P)$, where $P \in \mathbb{F}_2^{n\times n}$ is a permutation matrix.
1:  $Y = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \gamma_0 & \gamma_1 & \cdots & \gamma_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_0^{t-1} & \gamma_1^{t-1} & \cdots & \gamma_{n-1}^{t-1} \end{pmatrix}$
2:  $Z \leftarrow Diag_n(g(\gamma_0)^{-1}, g(\gamma_1)^{-1}, \ldots, g(\gamma_{n-1})^{-1})$
3:  $H \leftarrow YZ$
4:  Represent $H \in \mathbb{F}_{2^m}^{t\times n}$ as $H' \in \mathbb{F}_2^{mt\times n}$ by substituting each $h_{ij} \in \mathbb{F}_{2^m}$ with an element of $\mathbb{F}_2^{m\times 1}$.
5:  Compute a reduced row echelon form of $H'_r = [A|I_{mt}]$, where $A \in \mathbb{F}_2^{mt\times k}$
6:  $G \leftarrow [I_k|A^T]$
7:  Generate a random $S \in \mathbb{F}_2^{k\times k}$ of rank $k$
8:  Generate a random permutation matrix $P \in \mathbb{F}_2^{n\times n}$
9:  $G^{pub} \leftarrow SGP$
10: return $pk = (G^{pub}, t)$, $sk = (\mathbf{L}, g(X), P)$
```

2.3 Encryption

A plaintext is an arbitrary non-zero binary vector of length k, i.e. $\mathbf{m} \in \mathbb{F}_2^k \setminus \mathbf{0}$. A ciphertext $\mathbf{c} \in \mathbb{F}_2^n$ is the codeword of the code with generator matrix G^{pub}, which is distorted by a random noise of weight exactly t. We assume that the error vector $\mathbf{e} \in \mathbb{F}_2^n$ of weight t has been pre-computed in advance from some source of local randomness. Encryption is described in Algorithm 2.

Algorithm 2 Encryption

```
INPUT: $pk = (G^{pub} \in \mathbb{F}_2^{k\times n}, t \in \mathbb{N})$, $\mathbf{m} \in \mathbb{F}_2^k$, $\mathbf{e} \in \mathbb{F}_2^n$ with $w_H(\mathbf{e}) = t$
OUTPUT: $\mathbf{c} \in \mathbb{F}_2^n$
1: $\mathbf{y} \leftarrow \mathbf{m}G^{pub}$
2: $\mathbf{c} \leftarrow \mathbf{y} \oplus \mathbf{e}$
3: return $\mathbf{c}$
```

2.4 Decryption

The decryption procedure, described in Algorithm 3, uses decoding of the irreducible Goppa code as its main subroutine denoted by *Decode*$(\cdot)$. The latter uses Patterson's decoding algorithm [25], which is described in Algorithm 4. On input $\mathbf{c} \in \mathbb{F}_2^n$, it outputs either $\mathbf{y} = \mathbf{c} \oplus \mathbf{e}$, where $\mathbf{e} \in \mathbb{F}_2^n$ is the error vector computed by the decoding algorithm, or a special symbol "$\perp$" in the case of decoding error. In Step 1 of Algorithm 4, the values $(X - \gamma_i)^{-1} \mod g(X)$ are independent from the input, therefore we pre-comute them in order to improve performance.

Inversion over $\mathbb{F}_{2^m}[X]/g(X)$ in Steps 1 and 3 of Algorithm 4 is computed using the Extended Euclidean Algorithm. For instance in Step 1, we are looking for $Inv(X) \in \mathbb{F}_{2^m}[X]/g(X)$, such that $(X - \gamma_i)Inv(X) \equiv$

$1 \mod g(X)$, so that this algorithm is used to compute $h(X)$ such that $g(X)h(X)+(X-\gamma_i)Inv(X)=1$. In the same manner, this algorithm is used also in Step 3.

Square root over $\mathbb{F}_{2^m}[X]/g(X)$ is computed using Algorithm 5 – an adaptation of the square-root algorithm over $\mathbb{F}_{2^m}$ from [13, p. 136] (see also [4]). For Step 3, we pre-compute a lookup table for the square roots over $\mathbb{F}_{2^m}$, also we pre-compute the polynomials $R_i(X)$ in Step 2.

The so called *key equation* in the decoding algorithm is given by

$$\beta(X)\tau(X) \equiv \alpha(X) \mod g(X), \qquad (1)$$

where $\tau(X) \in \mathbb{F}_{2^m}[X]/g(X)$ is computed in Step 4 of Algorithm 4, $\alpha(X), \beta(X) \in \mathbb{F}_{2^m}[X]$, $\deg \alpha(X) \le \lfloor t/2 \rfloor$, $\deg \beta(X) \le \lfloor (t-1)/2 \rfloor$.

Algorithm 3 Decryption

```
INPUT: $sk = (\mathbf{L}, g(X), P)$ as described in Alg. 1;
$\mathbf{c} \in \mathbb{F}_2^n$
OUTPUT: $\mathbf{m} \in \mathbb{F}_2^k$ or $\perp$
1: $\mathbf{c} \leftarrow \mathbf{c}P^{-1}$
2: $\mathbf{y} = Decode(\mathbf{L}, g(X), \mathbf{c})$          /* See Alg. 4
3: if $\mathbf{y} = \perp$
       return $\perp$                        /* Decoding error
4: Compute $G_J \in \mathbb{F}_2^{k \times k}$ with $J \subseteq \{1, \dots, n\}$ s.t.
   $\mathrm{rank}(G_J) = k$
5: $\mathbf{m} \leftarrow \mathbf{y}_J (G_J)^{-1} S^{-1}$
6: return m
```

Algorithm 4 Error correction of binary irreducible Goppa codes

```
INPUT: $\mathbf{L} = (\gamma_0, \dots, \gamma_{n-1}) \in \mathbb{F}_{2^m}^n$,
$g(X) \in \mathbb{F}_{2^m}[X]$ with $\deg(g(X)) = t$
$\mathbf{c} = (c_0, \dots, c_{n-1}) \in \mathbb{F}_2^n$
OUTPUT: $\mathbf{y} \in \mathbb{F}_2^n$ or $\perp$
1: /* Computation of the syndrome
   $S_{\mathbf{c}}(X) \leftarrow \sum_{i=0}^{n-1} c_i (X-\gamma_i)^{-1} \in \mathbb{F}_{2^m}[X]/g(X)$
2: /* Check if c is a codeword
   if $S_{\mathbf{c}}(X) = 0$ then
       return c
3: /* Inverse over $\mathbb{F}_{2^m}[X]/g(X)$
   $T(X) \leftarrow S_{\mathbf{c}}^{-1}(X) \in \mathbb{F}_{2^m}[X]/g(X)$
4: /* Square root over $\mathbb{F}_{2^m}[X]/g(X)$     (See Alg. 5)
   $\tau(X) \leftarrow \sqrt{T(X)+X} \in \mathbb{F}_{2^m}[X]/g(X)$
5: /* Solving the key equation               (See Alg. 6)
   $\sigma(X) \leftarrow \alpha(X)^2 + X\beta(X)^2 \in \mathbb{F}_{2^m}[X]$,
   $\deg(\sigma(X)) \le t$; $\alpha(X), \beta(X)$ are defined in (1)
6: /* Root-finding for $\sigma(X)$                (See Alg. 7)
   $Roots \leftarrow$ Root-finding($\sigma(X)$) $\in \mathbb{F}_{2^m}^n$
7: for $i = 0$ to $n-1$ do
8:     for $j = 0$ to $n-1$ do
9:         if $Roots_i = \gamma_j$ then $\mathbf{e}_i \leftarrow 1$
                              else $\mathbf{e}_i \leftarrow 0$
10: /* If no roots were found, return an error
    if $\mathbf{e} = \mathbf{0}$ then
        return $\perp$
11: $\mathbf{y} \leftarrow \mathbf{c} \oplus \mathbf{e}$
12: return y
```

3 Environment and Implementation

3.1 Environment

For implementation and calculation of timing, we use a personal computer with the following environment: CPU AMD Phenom II X6 1090T 3.20 GHz, RAM 4.00 GB, OS Windows 7 Enterprise 64 bit, compiler Visual C++ 2010 Express Edition.

3.2 Implementation

In our implementation, we represent $\mathbb{F}_{2^m}$ as $\mathbb{F}_{2^m} \simeq \mathbb{F}_2[Y]/f(Y)$, where $f(Y)$ is an irreducible polynomial and $\deg(f(Y)) = m$.

For instance, for $\mathbb{F}_{2^m}$, we take $f(Y) = Y^{11}+Y^2+1$. The elements of $\mathbb{F}_{2^{11}}$ are represented as polynomials of degree at most 10 with coefficients in $\mathbb{F}_2$: $c_i \in \mathbb{F}_2$, $0 \le i \le 10$, $A = c_{10}Y^{10} + c_9Y^9 + \cdots + c_1Y^1 + c_0 \in \mathbb{F}_{2^{11}}$. We use a 32 bit integer to represent these polynomials as a bit sequence: $0 \cdots 0 c_{10} c_9 \cdots c_0$. The four basic arithmetical operations are defined by using bit operations.

The C++ class template provides us with generic programming. In our implementation, fields, field elements, polynomials and matrices are implemented by using class template.

In matrix operations over $\mathbb{F}_2$, we use the template specialization to override the default template implementation in order to realize fast computation. In particular, we use operations SSE2 and SSE4a. If an Intel CPU is used, one can also use SSE4 instead of SSE4a. The matrices are stored in memory column-by-column. Each column data are represented as a bit sequence and stored in a continuous memory block. Then, 128 bit SSE operations can be used to realize matrix computations. In 128 bit SSE operations, we use the **_mm_loadu_si128** instruction to load column data to a CPU register from memory and the **_mm_storeu_si128** instruction to store the resulting data to memory from the CPU register. The **_mm_xor_si128**, **_mm_and_si128** and **__popcnt** instructions are used in addition, multiplication and bitwise exclusive-or operations. In order to measure time, we use a performance counter.

This counter provides us with nano second resolution in our environment.

Algorithm 5 Computation of a square root over $\mathbb{F}_{2^m}[X]/g(X)$

```
INPUT: $Q(X) = q_0 + q_1 X + \ldots + q_{t-1}X^{t-1} \in \mathbb{F}_{2^m}[X]/g(X)$;
$g(X) \in \mathbb{F}_{2^m}[X]$ with $\deg(g(X)) = t$
OUTPUT: $\tau(X) = \sqrt{Q(X)} \mod g(X)$
1: $Sq(X) \leftarrow X^{2^{mt-1}} \mod g(X)$          /* $Sq(X) = \sqrt{X}$
2: for $i = 0$ to $\lfloor t/2 - 1 \rfloor$ do
       $R_i(X) = X^i Sq(X) \mod g(X)$
3: for $i = 0$ to $t - 1$ do
       $q'_i = q_i^{2^{m-1}}$          /* Computation over $\mathbb{F}_{2^m}$
4: $Q_{\text{even}}(X) \leftarrow \sum_{i=0}^{\lfloor (t-1)/2 \rfloor} q'_{2i} X^i$
5: $Q_{\text{odd}}(X) \leftarrow \sum_{i=0}^{\lfloor t/2-1 \rfloor} q'_{2i+1} R_i(X)$
6: $\tau(X) \leftarrow Q_{\text{even}}(X) + Q_{\text{odd}}(X)$
7: return $\tau(X)$
```

Algorithm 6 Solving the key equation

```
INPUT: $\tau(X) \in \mathbb{F}_{2^m}[X]/g(X)$
OUTPUT: $\sigma(X) \in \mathbb{F}_{2^m}[X]$ with $\deg(g(X)) = t$
1: /* Extended Euclidean Algorithm
2: $i \leftarrow 0$; $r_{-1}(X) \leftarrow g(X)$; $\alpha_{-1}(X) \leftarrow g(X)$;
   $r_0(X) \leftarrow \tau(X)$; $\alpha_0(X) \leftarrow \tau(X)$;
   $\beta_{-1}(X) \leftarrow 0$; $\beta_{-1}(X) \leftarrow 1$
3: while $\deg(r_i(X)) \geq \lfloor t/2 \rfloor$ do
4:     $i \leftarrow i + 1$
5:     By polynomial long division $\frac{r_{i-2}(X)}{r_{i-1}(X)}$,
       compute $q_i(X)$ and $r_i(X)$ s.t.
       $r_{i-2}(X) = q_i(X) r_{i-1}(X) + r_i(X)$
       and $\deg(r_i(X)) < \deg(r_{i-1}(X))$
6:     $\beta_i(X) \leftarrow \beta_{i-2}(X) + q_i(X)\beta_{i-1}(X)$
7:     $\alpha_i(X) \leftarrow r_i(X)$
8: end while
9: $\sigma(X) \leftarrow \alpha_i(X)^2 + X\beta_i(X)^2$
10: return $\sigma(X)$
```

4 Timing

As pointed out in [9, Sec. 8.3]: "There is no simple criterion for the choice of t with respect to n". Moreover, since implementations of the McEliece PKC appeared only in the recent few years, there are no commonly accepted benchmark parameters. The successful attack against parameters proposed originally by McEliece [20] ($n = 1024, k = 524, t = 50$) has been carried out by Bernstein et al [2] in about $2^{60.5}$ operations, and the challenge ciphertext was decrypted. However, no such attacks against the parameter sets with $n = 2048$ has been devised fo far. The timing data, which are presented in this section, are an average over 100 messages encrypted and decrypted on the same public/secret key pair for each parameter set. The code support and the irreducible Goppa polynomial for the key pair were generated at random. We note that Steps 4-5 of the decryption algorithm (Alg. 3) have not been implemented, therefore the timing for these steps is not provided.

Algorithm 7 Root-finding Algorithm

```
INPUT: $f(X) \in \mathbb{F}_{2^m}[X]$, $\deg(f(X)) \leq t$
OUTPUT: Roots of $f(X) \in \mathbb{F}_{2^m}[X]$
1: $d \leftarrow m$
2: $k \leftarrow \deg(f(X))$
3: Normalize $f(X)$          /* $f(X)$ becomes monic
4: if $k = 1$ then
5:     return free term of $f(X)$          /* Output a root
6: else
7:     $T(X) \leftarrow X^2 + X$
8:     while true do
9:         $U(X) \leftarrow T(X) + T(X)^2 + T(X)^4 + \cdots + T(X)^{2^{d-1}} \mod f(X)$
10:        $B(X) \leftarrow \gcd(f(X), U(X))$
11:        Normalize $B(X)$
12:        if $\deg(B(X)) = 0$ or $\deg(B(X)) = \deg(f(X))$ or $B(X) = f(X)$ then
13:            $c_0, c_1 \leftarrow$ random element of $\mathbb{F}_{2^m}$
14:            /* Update $T(X)$ for the next loop
               $T(X) \leftarrow X^2 + c_1 X + c_0$
15:            continue          /* Go to the next loop
16:        else
17:            $S(X) = f(X)/B(X)$
18:            recursive call:
                   goto 2 with $f(X) \leftarrow B(X)$
19:            recursive call:
                   goto 2 with $f(X) \leftarrow S(X)$
20:        end if
21:    end while
22: end if
```

First, we report timing for the parameters $n = 2048$ ($m = 11$) and $t = 31, 51, 81$. The results are summarized in Table 1. The corresponding timing for the key generation step (Algorithm 1) is 29.85, 46.76 and 96.43 seconds, respectively. These data include the timing for all pre-computations: inverses for the syndrome, the square-root lookup table, etc.

In Table 1 and later, we compute the equivalent length of AES key corresponding to the given McEliece PKC parameters using the approximate bound from [9, p. 196], omitting a multiplicative factor polynomial in n:

$$O(n^3)2^{-t\log_2(1-k/n)}. \qquad (2)$$

In Table 1 with $n = 2048$, we choose the first parameter set with $t = 31$ because it provides 80-bit security, a widely accepted (de facto) standard for the minimal level of security. Increasing t to 51 raises the security to 95 bits, but the decryption time becomes about 2.4 times lower. A further increase of t to 81 raises the decryption time about 2.5 times, compared to that of with $t = 51$, but security is only raised by 2 bits. This shows that if the system designer is flexible with respect to security requirements, she can substantially win in performance, sacrificing only a little in terms of security. Note that this conclusion is only preliminary,

Table 1: Timing for $n = 2048$ ($m = 11$), in msec.
* Security level corresponding to the given parameters.

Algorithm	**Running Time**		
	$t = 31$	$t = 51$	$t = 81$
	80 bit*	95 bit*	97 bit*
Encryption			
Algorithm 2 (total)	**1.06**	**1.01**	**1.00**
Matrix multip.: St. 1	1.02	0.97	0.96
Error vector: St. 2	0.04	0.04	0.04
Decoding			
Algorithm 4 (total)	**116.41**	**278.20**	**702.82**
Syndrome: Step 1	18.31	29.54	48.12
Inverse: Step 3	11.78	30.33	76.86
Square root: Step 4	10.39	43.87	179.52
Key equation: Alg. 6	4.56	11.76	30.21
Root finding: Alg. 7	71.37	162.70	368.11

since the contribution of solving the system of linear equations in Steps 4-5 of decryption (Algorithm 3) have not yet been evaluated. The encryption time is about 1 millisecond for all $t = 31, 51, 81$. In fact, the encryption time is decreasing from $t = 31$ to $t = 81$ by a few hundredth of a millisecond, but we explain it as an effect of noise. The same effect is observed in Table 2 as well.

Next, we present timing for the parameters $n = 4096$ ($m = 12$) and $t = 41, 62, 101$ in Table 2. The corresponding timing for the key generation step (Algorithm 1) is 3 minutes 17 seconds, 5 minutes 15 seconds and 9 minutes 12 seconds, respectively. As before, these data include the timing for all pre-computations.

Table 2: Timing for $n = 4096$ ($m = 12$), in msec.
* Security level corresponding to the given parameters.

Algorithm	**Running Time**		
	$t = 41$	$t = 62$	$t = 101$
	130 bit*	160 bit*	190 bit*
Encryption			
Alg. 2 (total)	**2.42**	**2.53**	**2.34**
Multiplic.: St. 1	2.34	2.45	2.26
Error vector: St. 2	0.08	0.08	0.08
Decoding			
Alg. 4 (total)	**235.67**	**500.91**	**1300.61**
Syndrome: Step 1	50.31	77.48	122.88
Inverse: Step 3	21.74	47.55	126.14
Square root: Step 4	24.84	86.73	367.33
Key eq.: Alg. 6	8.14	18.27	49.61
Root finding: Alg. 7	130.64	270.88	634.65

In Table 2, we see that the decoding time for ($n = 4096, t = 41$) is close to that of ($n = 2048, t = 51$), and encryption is 2 times slower for the former. However, the security level of the set ($n = 4096, t = 41$) is 130 bits, compared to only 95 bits for that ($n = 2048, t = 51$). This shows that increasing n, but decreasing t is one of the tradeoffs for obtaining higher security.

In accordance with the trends shown in Table 1, one can see also from Table 1 that increasing security by 30 bits from 130 bits with ($n = 4096, t = 41$) to 160 bits with ($n = 4096, t = 62$) comes for the price of increasing the decoding time by about 2.12 times. At the same time, the next upgrade of secuirty to 190 bits with ($n = 4096, t = 101$) will result in increasing of the decoding time by about 2.6 times.

The tendency in Tables 1 and 2 leads us to a preliminary conclusion that for every n, there exists a range of t, which allows us to efficiently realize a trade-off between security level and decoding speed. For the lower bound, t must provide the minumum required security level (say 80 bit). For the uppr bound, an increase of t must provide a substantial increase in security level according to the function (2), yet not to cause a prohibitively large decoding time.

For all the aforementioned parameters, the key pair sizes as well the size of the pre-computed data are provided in Table 3. Note that these are theoretical data, meaning that an element of $\mathbb{F}_2$ is counted as 1 bit, the element of $\mathbb{F}_{2^{11}}$ as 11 bits and so on. In practice, the memory taken by these data is typically larger.

4.1 Discussion on Previous Works

As mentioned in the introduction, our implementation is not competitive with respect to the existing one by Strenzke [27]. He reports [27, Table 2] encryption in 1.2 millisecond and decryption in 1.6 millisecond for the parameters ($n = 2048, t = 50$) on the personal computer with Intel Core Duo T7300 2GHz running Linux with kernel version 2.6.24, the implementation compiled with GCC-4.1.3, optimization level O2. Our timing for ($n = 2048, t = 51$) is 1.01 millisecond for encryption and 278.2 milliseconds for decryption, while we use 3.2 GHz CPU.

Döring [7, Ch. 5] in his thesis implements the IND-CCA2 secure conversion (for the McEliece PKC) by Kobara and Imai [18]. His timing is obtained on a personal computer with a Pentium M 1.6 GHz CPU, 2 GB of RAM, with Microsoft Windows XP. The code is compiled with JDK 1.3 and run under JRE 1.6. Döring reports [7, Tab. 2.5] for parameters ($n = 2048, t = 50$), key generation in 3.8 seconds, encryption in 3.6 milliseconds and decryption in 40.6 milliseconds.

Biswas in his thesis [4] implements the modified version of the McEliece PKC called HyMES [5], which was proposed by Biswas and Sendrier. The environment for timing calculation was a personal computer with Intel Core 2 processor with dual core, running a 32 bits operating system and a single core. The C program was compiled with icc Intel compiler with the options "-g -static -O -ipo -xP". Biswas [4, Table 2.1] reports encryption in 223 cycles/byte and decryption in 2577 cycles/byte for the parame-

Table 3: Size of keys and pre-computed data

	$m = 11$, $n = 2048$			$m = 12$, $n = 4096$		
k	1707	1487	1157	3604	3352	2884
t	31	51	81	41	62	101
Public key pk	427 kB	372 kB	289 kB	1.76 MB	1.64 MB	1.41 MB
Secret key sk	5.54 kB	5.57 kB	5.61 kB	12.06 kB	12.09 kB	12.14 kB
Pre-computed data	86 kB	142 kB	227 kB	247 kB	375 kB	614 kB

ters ($n = 2048, t = 40$). A straight forward computations for the 3.2 GHz CPU (our case) would result in 0.02 milliseconds for encryption and 0.2 milliseconds for decryption. This would be a remarkable performance, but since we are not sure about the precise meaning of "cycles/byte" in [4], we will refrain from any futher comments.

5 Conclusion

We implemented postquantum McEliece PKC [20]. In the decryption algorithm, we only implemented the decoding part. Timing estimations show that our work must be optimized in order to be competitive with existing results.

Future works, apart from completion of the current implementation, include realization of compact keys [21], [17] and IND-CCA2 secure conversion [18].

References

[1] E. Berlekamp, R. McEliece, and H. van Tilborg, "On the inherent intractability of certain coding problems", IEEE Trans. on Inf. Theory 24, pp. 384-386, 1978.

[2] D.J. Bernstein, T. Lange, C. Peters, "Attacking and Defending the McEliece Cryptosystem", PQCrypto 2008, pp. 31-46, 2008.

[3] D.J. Bernstein, T. Lange and C. Peters, "Smaller Decoding Exponents: Ball-Collision Decoding", CRYPTO 2011, pp. 743-760, 2011.

[4] B. Biswas, "Implementational aspects of code-based cryptography", Ph.D. thesis, Ecole Polytechnique, 2010. Available at: `http://pastel.archives-ouvertes.fr/docs/00/52/30/07/PDF/thesis.pdf`

[5] B. Biswas, N. Sendrier, "Hybrid McEliece", Available at: `http://www-rocq.inria.fr/secret/CBCrypto/index.php?pg=hymes`

[6] H. Cohen, "A course in computational algebraic number theory", Graduate texts in mathematics, Springer-Verlag, 1993.

[7] M. Döring, "On the Theory and Practice of Quantum-Immune Cryptography", Ph.D. thesis, Technical University of Darmstadt, 2008.

[8] T. Eisenbarth, T. Güeysu, S. Heyse, C. Paar, "MicroEliece: McEliece for Embedded Devices", CHES 2009, pp. 49-64, 2009.

[9] D. Engelbert, R. Overbeck and A. Schmidt: A Summary of McEliece-Type Cryptosystems and their Security, Journal of Mathematical Cryptology, vol. 1 , pp. 151-199, Walter de Gruyter, 2007.

[10] M. Finiasz, N. Sendrier, "Security Bounds for the Design of Code-Based Cryptosystems", ASIACRYPT 2009, pp. 88-105, 2009.

[11] V. D. Goppa, "A new class of linear error-correcting code" (in. Russian), Probl. Peredach. Inform., vol. 6, pp. 24-30, Sept. 1970.

[12] S. Heyse, "Low-Reiter: Niederreiter Encryption Scheme for Embedded Microcontrollers", PQCrypto 2010, pp. 165-181, 2010.

[13] D.R. Hankerson, S.A. Vanstone, and A.J. Menezes, "Guide to elliptic curve cryptography", Springer professional computing, Springer, 2004.

[14] S. Heyse, A. Moradi, C. Paar, "Practical Power Analysis Attacks on Software Implementations of McEliece", PQCrypto 2010, pp. 108-125, 2010.

[15] G. Hoffmann, "Implementation of McEliece using quasi-dyadic Goppa codes", B.Sc. thesis, Technical University of Darmstadt, 2011. Available at: `http://www.cdc.informatik.tu-darmstadt.de/reports/reports/Gerhard_Hoffmann.bachelor.pdf`

[16] T. Howenga, "Efficient Implementation of the McEliece Cryptosystem on Graphics Processing Units", M.Sc. thesis, Ruhr-University Bochum, Germany, 2009.

[17] K. Kobara, "Flexible Quasi-Dyadic Code-Based Public-Key Encryption and Signature", Cryptology ePrint Archive: Report 2009/635, Available at: `http://eprint.iacr.org/2009/635.pdf`

[18] K. Kobara, H. Imai, "Semantically Secure McEliece Public-Key Cryptosystems - Conversions for McEliece PKC -", PKC 2001, pp.19–35, 2001.

[19] R. Lidl, H. Niederreiter, "Introduction to finite fields and their applications", Cambridge University Press, 1986.

[20] R.J. McEliece, "A Public-Key Cryptosystem Based on Algebraic Coding Theory," Deep Space Network Progress Rep., 1978.

[21] R. Misoczki, P.S.L.M. Barreto, "Compact McEliece Keys from Goppa Codes", SAC 2009, pp. 376-392, 2009.

[22] Microsoft Developer Network, C/C++ Languages, Compiler Intrinsics. Available at: `http://msdn.microsoft.com/en-us/library/26td21ds.aspx`

[23] F. J. MacWilliams and N. J. A. Sloane, "The Theory of Error-Correcting Codes", 7. edn. North-Holland Amsterdam, 1992.

[24] R. Overbeck, "An Analysis of Side Channels in the McEliece PKC (2008)", Presentation available at: `https://www.cosic.esat.kuleuven.be/natoarw/slidesparticipants/Overbeck_slides_nato08.pdf`

[25] N. J. Patterson, "The algebraic decoding of Goppa codes", IEEE Trans. on Inf. Theory, Vol. IT-21, pp. 203-207, 1975.

[26] N. Sendrier, "On the security of the McEliece public-key cryptosystem", Information, Coding and Mathematics – Proceedings of Workshop honoring Prof. Bob McEliece on his 60th birthday, pp. 141-163, Kluwer, 2002.

[27] F. Strenzke, "A Smart Card Implementation of the McEliece PKC", WISTP 2010, pp. 47-59, 2010.