A Smart Card Implementation of the McEliece PKC

Falko Strenzke[1,2]

[1] FlexSecure GmbH, Germany[⋆]
strenzke@flexsecure.de
[2] Cryptography and Computeralgebra, Department of Computer Science, Technische Universität Darmstadt, Germany

Abstract. In this paper we present a smart card implementation of the quantum computer resistant McEliece Public Key Cryptosystem (PKC) on an Infineon SLE76 chip. We describe the main features of the implementation which focuses on performance optimization. We give the resource demands and timings for two sets of security parameters, the higher one being in the secure domain. The timings suggest the usability of the implementation for certain real world applications.

Keywords: Public key encryption scheme, code-based cryptography, post quantum cryptography, smart card, implementation.

1 Introduction

Current public key schemes like RSA and elliptic curve based cryptosystems depend on the complex mathematical problems of integer factorization and the calculation of discrete logarithms [1,2,3,4]. These systems are known to be vulnerable against so-called quantum algorithms which could be run efficiently on quantum computers [5,6,7]. Today, practically useful quantum computers have not been build, but they are subject to intensive research. It is virtually impossible to predict how long it will take scientists to construct a quantum computer of sufficient potency to break today's cryptographic schemes. But once this is the case, new classes of cryptographic schemes will be needed to furthermore guarantee data security. These new types of cryptographic schemes we refer to as quantum computer resistant cryptographic schemes or post quantum cryptography. Examples for theses types of algorithms are the hash-based cryptography, such as the Merkle signature scheme [8,9], and code-based cryptography like the McEliece PKC [10,11], which is the subject of this paper.

While in principle it would suffice to switch to a quantum computer resistant signature scheme just when quantum computers become an actual threat, matters stand different for encryption schemes. Data that is encrypted today and sent through a public channel might be recorded and stored by an attacker. Then, once quantum computers are available to him, he is able to decrypt his

[⋆] A part of the work of F. Strenzke was done at[2].

recorded ciphertexts. This scenario shows the importance and urgency of taking precautions against the threat that quantum computers pose for today's public key cryptosystems.

In this paper we present an implementation of the McEliece PKC on a smart card. In Section 2 we describe the key generation as well as the encryption and decryption operations of the McEliece PKC. A description of the hardware platform and the most basic software design choices are given in Section 3. In Section 4 we show timings for the encryption and decryption operations using realistic security parameters. The timings are in the magnitude of seconds and thus imply the usability of the implementation for certain purposes. In Section 5 we give the conclusion and consider possible future improvements to the implementation.

2 Preliminaries

In the following we will give a brief definition of classical Goppa codes and the McEliece PKC. In this section we assume that the reader is familiar with the basics of error correcting codes. We use the notation given e.g. in [12].

2.1 Classical Goppa Codes

Goppa codes [13] are a class of linear error correcting codes. The McEliece PKC makes use of irreducible binary Goppa codes, so we will restrict ourselves to this subclass.

Definition 1. *Let the polynomial*

$$g(X) = \sum_{i=0}^{t} g_i X^i \in \mathbb{F}_{2^m}[X] \tag{1}$$

be monic and irreducible over $\mathbb{F}_{2^m}[X]$*, and let* m*,* t *be positive integers. Then* $g(X)$ *is called a* Goppa polynomial *(for an irreducible binary Goppa code).*

Then an irreducible binary Goppa code is defined as

$$\mathcal{G}(\mathbb{F}_{2^m}, g(X)) = \{\boldsymbol{c} \in \mathbb{F}_2^n | S_{\boldsymbol{c}}(X) := \sum_{i=0}^{n-1} \frac{c_i}{X - \gamma_i} = 0 \bmod g(X)\} \tag{2}$$

where $n \leq 2^m$*,* $S_{\boldsymbol{c}}(X)$ *is the syndrome of* $\boldsymbol{c}$*, the* γ_i*,* $i = 0, \ldots, n-1$ *are pairwise distinct elements of* $\mathbb{F}_{2^m}$*, and* c_i *are the entries of the vector* $\boldsymbol{c}$*.*

The code defined in such way has length n, dimension $k = n - mt$ and can correct up to t errors. Please note that our implementation only allows lengths $n = 2^m$. The canonical check matrix $\mathbf{H}$ for $\mathcal{G}(\mathbb{F}_{2^m}, g(X))$ can be computed from the syndrome equation and is given in Appendix A.

2.2 The McEliece PKC

The McEliece PKC is named after its inventor [10]. It is a public key encryption scheme based on general coding theory. Specifically, the McEliece PKC uses Goppa codes. The strongest known attack against this scheme is based on solving the NP-hard decoding problem, and no quantum algorithm has been proposed which increases the efficiency of this attack [14].

In the following, we will give a brief description of the individual algorithms for key generation, encryption and decryption, without presenting the mathematical foundations behind the scheme or the consideration of its security. For these considerations, the reader is referred to [15].

It was shown that the original McEliece PKC is vulnerable against chosen-ciphertext attacks, see [15] for an overview. However, this problem can be solved by incorporating a CCA2-conversion in the scheme. A number of such conversions have been proposed for the McEliece PKC [11]. The conversion we are using in our implementation is described in Section 2.6.

Parameters of the McEliece PKC. The security parameters $m \in \mathbb{N}$ and $t \in \mathbb{N}$ with $t \ll 2^m$ have to be chosen in order to set up a McEliece PKC. An example for secure values would be $m = 11$, $t = 50$. These values can be derived from the considerations given in [16] and [17].

2.3 McEliece Key Generation

The private key. The private key consists of two parts. The first part of the secret key in the McEliece PKC is a Goppa polynomial $g(X)$ of degree t over $\mathbb{F}_{2^m}$ according to Definition 1, with random coefficients. The second part is a randomly created $n \times n$ permutation matrix $\mathbf{P}$.

The public key. The public key is generated from the secret key as follows. First, compute $\mathbf{H}$ as the parity check matrix corresponding to $g(X)$. Then take $\mathbf{G}^{\text{pub}} = [\mathbb{I}_k \mid \mathbf{R}]$ as the generator in systematic form corresponding to the parity check matrix $\mathbf{HP}^\top$ (refer to Appendix A for the creation of the parity check matrix and the generator of a Goppa code). Please note that choosing the generator in systematic form would be a security problem if the McEliece PKC was used without a CCA2-conversion.

2.4 McEliece Encryption

Assume Alice wants to encrypt a message $\boldsymbol{m} \in \mathbb{F}_2^k$. Firstly, she has to create a random binary vector $\boldsymbol{e}$ of length n and Hamming weight $\text{wt}(\boldsymbol{e}) = t$. Then she computes the ciphertext $\boldsymbol{z} = \boldsymbol{m}\mathbf{G}^{\text{pub}} \oplus \boldsymbol{e}$.

2.5 McEliece Decryption

In order to decrypt the ciphertext, Bob computes $\boldsymbol{z}' = \boldsymbol{z}\mathbf{P}^{-1}$. He then computes the syndrome $S_{\boldsymbol{z}'} = \boldsymbol{z}'\mathbf{H}^T$. Afterwards he applies error correction by executing an error correction algorithm, which receives as its input the syndrome and the permuted distorted codeword $\boldsymbol{z}'$. It outputs the so called error locator polynomial defined as

$$\sigma_{\boldsymbol{e}'}(X) = \prod_{j \in \mathcal{T}_{\boldsymbol{e}'}} (X - \gamma_j) \in \mathbb{F}_{2^m}[X],$$

where $\mathcal{T}_{\boldsymbol{e}'} = \{i | e'_i = 1\}$ and $\boldsymbol{e}'$ is the error vector of the permuted distorted code word $\boldsymbol{z}'$. Once the error locator polynomial is known, the permuted error vector is computed as

$$\boldsymbol{e}' = (\sigma_{\boldsymbol{e}'}(\gamma_0), \sigma_{\boldsymbol{e}'}(\gamma_1), \cdots, \sigma_{\boldsymbol{e}'}(\gamma_{n-1})) \oplus (1, 1, \cdots, 1),$$

i.e. $e'_i = 1$ if $\sigma_{\boldsymbol{e}'}(\gamma_i) = 0$ and $e'_i = 0$ otherwise. The error vector then is found by undoing the permutation: $\boldsymbol{e} = \boldsymbol{e}'\mathbf{P}$. Then the message is recovered as the first k bits of $\boldsymbol{z} \oplus \boldsymbol{e}$.

In our implementation, we use the Patterson Algorithm [18] as error correction algorithm.

The Patterson Algorithm. The Patterson Algorithm. is an efficient algorithm for the determination of the error locator polynomial. We will give a brief description of the algorithm without any proofs.

The algorithm uses the fact that the error locator polynomial can be written as

$$\sigma_e(X) = \alpha^2(X) + X\beta^2(X). \tag{3}$$

Defining

$$\tau(X) = \sqrt{S_{\boldsymbol{z}}^{-1}(X) + X} \bmod g(X)\,, \tag{4}$$

with $S_{\boldsymbol{z}}(X)$ being the syndrome of the distorted code word $\boldsymbol{z}$, the following equation holds:

$$\beta(X)\tau(X) = \alpha(X) \bmod g(X) \tag{5}$$

Then, assuming that no more than t errors occurred, Equation 5 can be solved by applying the Euclidean algorithm with a breaking condition concerning the degree of the remainder [15]. Specifically, the remainder in the last step is taken as $\alpha(X)$ and the breaking condition is $\deg(\alpha(X)) \leq \lfloor \frac{t}{2} \rfloor$. It can be shown that then, $\deg(\beta(X)) \leq \lfloor \frac{t-1}{2} \rfloor$.

Thus, once $\alpha(X)$ and $\beta(X)$ are determined, the error locator polynomial $\sigma_{\boldsymbol{e}}$ is known.

2.6 CCA2-Conversion

As mentioned in Section 2.2, the original McEliece PKC needs to be extended by a CCA2-conversion to achieve security against chosen-ciphertext attacks. The conversion we are using in our implementation is introduced in [19] and was designed with respect to optimized computation time and side channel resistance. A security proof for this conversion will be given elsewhere. Note that in the implementation the CCA2-conversion is easily exchangeable.

In the following, we will use

$$(\boldsymbol{z}, \boldsymbol{e}) \leftarrow \mathcal{E}_{\mathbf{G}^{\mathrm{pub}}}(\boldsymbol{m})$$

to denote the McEliece encryption of the message $\boldsymbol{m}$ to the ciphertext $\boldsymbol{z}$ using the public key $\mathbf{G}^{\mathrm{pub}}$, as depicted in Section 2.4. The error vector $\boldsymbol{e}$ is also modeled as an output of the encryption function, since it is needed in the CCA2-conversion which encapsulates the McEliece encryption. The same applies to the decryption

$$(\boldsymbol{m}, \boldsymbol{e}) \leftarrow \mathcal{D}_{(P,g(X))}(\boldsymbol{z}),$$

i.e. here $\boldsymbol{e}$ is also an output of the algorithm.

The conversion makes use of a hash function $H()$ which outputs a bit vector of length l. In the implementation, we are using SHA256, so $l = 256$. Furthermore, by $\|$ we denote concatenation.

Note that in Algorithm 1 and 2, the ciphertext part $\boldsymbol{z}_1$ has a bit length equal to the parameter n of the McEliece PKC, whereas $\boldsymbol{z}_2$ and $\boldsymbol{z}_3$ are of bit length l.

Algorithm 1. McEliece - CCA2 secure encryption

```
Require: message $\boldsymbol{m} \in \mathbb{F}_2^l$, public key $\mathbf{G}^{\mathrm{pub}}$
Ensure: ciphertext $\boldsymbol{z} \in \mathbb{F}_2^{n+2l}$
  $\boldsymbol{u}_1 \leftarrow$ random $(k-l)$-bit string.
  $\boldsymbol{u}_2 \leftarrow$ random $l$-bit string.
  $(\boldsymbol{z}_1, \boldsymbol{e}) \leftarrow \mathcal{E}_{\mathbf{G}^{\mathrm{pub}}}(\boldsymbol{u}_1 \| H(m \| \boldsymbol{u}_2))$
  $z \leftarrow \left( \boldsymbol{z}_1 \, \| \, \underbrace{(H(\boldsymbol{u}_1) \oplus m)}_{\boldsymbol{z}_2} \, \| \, \underbrace{(\boldsymbol{u}_2 \oplus H(e))}_{\boldsymbol{z}_3} \right)$
```

Algorithm 2. McEliece - CCA2 secure decryption

```
Require: ciphertext $\boldsymbol{z} = (\boldsymbol{z}_1, \boldsymbol{z}_2, \boldsymbol{z}_3) \in \mathbb{F}_2^{n+2l}$, secret key $(P, g(X))$
Ensure: decrypted message $\boldsymbol{m} \in \mathbb{F}_2^l$
  $(\boldsymbol{w}, \boldsymbol{e}) \leftarrow \mathcal{D}_{(P,g(X))}(\boldsymbol{z}_1)$
  $\boldsymbol{r} \leftarrow$ the first $k-l$ bits of $\boldsymbol{w}$
  $\boldsymbol{h} \leftarrow$ the bits at $k-l+1, \cdots, k$ of $\boldsymbol{w}$.
  $\boldsymbol{m} \leftarrow \boldsymbol{z}_2 \oplus H(\boldsymbol{r})$
  if $\boldsymbol{h} = H(\boldsymbol{m} \| (H(\boldsymbol{e}) \oplus \boldsymbol{z}_3))$ then
    return $\boldsymbol{m}$
  else
    return error
  end if
```

3 Features of the Implementation

In this Section we outline the most basic features of our software implementation and the hardware platform we are using.

3.1 The Hardware Platform

As hardware platform, we use an SLE76CF5120P controller out of the SLE76-family [20] by Infineon Technologies AG. It features a 16 bit CPU based on the 80251 architecture. It has a clock rate of 33 MHz and provides 12 kByte of RAM. It es equipped with 504 kByte of non-volatile memory (NVM, i.e. flash memory). It also features a unified data and code cache of 1 kByte.

3.2 Design of the Software

As we did not use a preexisting smart card operating system, we had to implement the basic functions for memory management and I/O. This encompasses sending and receiving data via the serial interface, command processing and management of the heap memory. Despite from this, a large number of mathematical routines is needed for the encryption and decryption algorithms of the McEliece PKC. The source code is written in the C programming language but is strongly object oriented. In the following we give a brief overview of the most important mathematical objects modeled in the code.

With respect to the prototypic nature of our implementation, we chose to optimize with regard to execution time, not memory usage. This is based on the following considerations. In a real world application the available RAM and NVM would be firmly determined by the actual hardware and OS platform. There, the time-memory tradeoffs arising in the implementation would have to be shifted towards reducing memory usage until the limitations are fulfilled. Without being given any concrete limitations, it seems more useful to show the so far best achievable performance.

The field $\mathbb{F}_{2^m}$. The Galois Field implementation is taken from the open source McEliece implementation [21], results concerning this implementation are given in [22]. This implementation uses lookup tables for the computation of exponentiations and logarithms of elements. These in turn are used to build most other operations in $\mathbb{F}_{2^m}$. All of these operations are implemented as preprocessor macros. The Galois Field implementation determines the speed of the decryption operation to a large extent, as it realizes the lowest level of implementation for all polynomial operations.

Polynomials over $\mathbb{F}_{2^m}$. In order to minimize execution time, each coefficient is implemented as a two byte word. This in turn means that for our actual choices of the parameter $m = 10$ and $m = 11$ (see Section 4.1), a considerable number of bits in each word is unused.

Permutations. Permutations are implemented as lookup tables. Again, each entry is two bytes wide. Accordingly, a number of bits is in each word remains unused. Note that there is no need to store the inverse of the permutation. A function to apply the inverse of the permutation is easily implemented.

Matrices over $\mathbb{F}_2$. Clearly, no memory-time tradeoff choices arise with respect to binary matrices. Matrices are stored row-wise. It is, however, important to realize that in the binary case, by using the efficient "vector $\times$ matrix" type multiplication, the Hamming weight of the vector is leaked through the running time of the operation. This is not a problem in the decryption operation, as according to Section 2.5 the only operation preceding the computation of the syndrome vector (by multiplying $\boldsymbol{z}'$ by the parity check matrix) is the application of the permutation $\mathbf{P}$. Note that the application of the permutation leaves the Hamming weight of the vector invariant. Accordingly, in our implementation, we use this type of matrix-vector multiplication for syndrome computation.

In case of the encryption operation (Section 2.4) however, the message vector is multiplied by the generator matrix. Here we have to use a "matrix $\times$ vector" type multiplication in order not to leak the Hamming weight of the message through a timing side channel. Accordingly, the public key is stored as $\mathbf{G}^{\mathrm{pub}^T}$ in the implementation.

Private Key. As shown in Section 2.3, the private key consists of the Goppa polynomial $g(X)$ and the permutation $\mathbf{P}$. However, to allow for an efficient decryption operation, further precomputed objects have to be available.

Parity Check Matrix. In order to perform efficient computation of the syndrome (see Section 2.5) the parity check matrix $\mathbf{H}$ has to be stored. It makes up the major part part of the private key, as is shown in section 4.3.

Square root matrix. Furthermore, as the Patterson Algorithm involves the computation of a square root in $\mathbb{F}_{2^m}[X]/g(X)$ (Equation 4), it is helpful to store a so called square root matrix [23,24] as part of the private key. With the help of this matrix, computing square roots in $\mathbb{F}_{2^m}[X]/g(X)$ is split into a matrix multiplication and computing square roots in $\mathbb{F}_{2^m}$. In this way, performance is greatly enhanced. The square root matrix is computed during the key generation in the following way. First, the squaring matrix Q for squarings in $\mathbb{F}_{2^m}[X]/g(X)$ is generated as a $t \times t$ matrix with coefficients in $\mathbb{F}_{2^m}$ as follows: Generate the i-th column as as $X^{2i} \bmod g(X)$ for $i \in \{0, \ldots t-1\}$, where each coefficient goes into one row.

$$Q = \begin{pmatrix} \gamma_1 & \gamma_0 & \gamma_0 & \gamma_0 & \cdots & \gamma_0 & q_0^{(0)} & \cdots & q_0^{(s)} \\ \gamma_0 & \gamma_0 & \gamma_0 & \gamma_0 & \cdots & \gamma_0 & q_1^{(0)} & \cdots & q_1^{(s)} \\ \gamma_0 & \gamma_1 & \gamma_0 & \gamma_0 & \cdots & \gamma_0 & q_2^{(0)} & \cdots & q_2^{(s)} \\ \gamma_0 & \gamma_0 & \gamma_0 & \gamma_0 & \cdots & \gamma_0 & q_3^{(0)} & \cdots & q_3^{(s)} \\ \gamma_0 & \gamma_0 & \gamma_1 & \gamma_0 & \cdots & \gamma_0 & q_4^{(0)} & \cdots & q_4^{(s)} \\ \gamma_0 & \gamma_0 & \gamma_0 & \gamma_0 & \cdots & \gamma_0 & q_5^{(0)} & \cdots & q_5^{(s)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \gamma_0 & \gamma_0 & \gamma_0 & \gamma_0 & \cdots & \gamma_1 & q_{t-1}^{(0)} & \cdots & q_{t-1}^{(s)} \end{pmatrix} \tag{6}$$

Here, γ_0 and γ_1 represent the neutral element regarding addition and multiplication in $\mathbb{F}_{2^m}$, respectively. The buildup of the matrix Q is as follows. The first

$\lceil t/2 - 1 \rceil$ columns simply represent the mapping $\gamma_1 \rightarrow \gamma_1$, $X \rightarrow X^2$, $X^2 \rightarrow X^4$, etc. and thus are independent of $g(X)$. Once the squaring causes a polynomial of degree t or higher, the reduction by $g(X)$ must be carried out, causing the entries $q_b^{(a)}$ to depend on the Goppa Polynomial. Here, the subscript and superscript of q simply indicate the rows and columns of the submatrix dependent on $g(X)$ and thus $s = t - 1 - \lceil t/2 - 1 \rceil$.

Note that the matrix depicted in Equation 6 is for an even value of the parameter t. In the case of t odd, the column containing the γ_1 in the last row would not exist.

The squaring of a polynomial with coefficients $\alpha = (\alpha_0, \alpha_1, \dots, \alpha_{t-1})$ can now be carried out by computing

$$\alpha^2 = Q\alpha',$$

where $\alpha' = (\alpha_0^2, \alpha_1^2, \dots, \alpha_{t-1}^2)$.

The desired square root matrix is found as Q^{-1}. Taking the square root of a polynomial β then amounts to compute $\beta' = Q^{-1}\beta$ and $\sqrt{\beta} = (\sqrt{\beta'_0}, \sqrt{\beta'_1}, \dots, \sqrt{\beta'_{t-1}})$.

4 Timings and Resource Usage

In this section we give timings and resource demands of our implementation for two sets of security parameters.

4.1 Parameter Sets

The parameters m and t of the McEliece PKC determine the security of the scheme and its resource demands. We tested our implementation with two sets of parameters, shown in the table below. The bit security is given with respect to the attack given in [17], which is to the best of our knowledge the strongest known attack today. Note that the message size is determined merely by the length of the output of the hash function used in the CCA2-conversion. In our implementation, this is always SHA256 (see Section 2.6).

Table 1. Security parameter sets for the McEliece PKC

m,t	**security bits**	**message size in byte**	**ciphertext size in byte**
10,40	62	32	$2 \cdot 32 + 128 =$ 192
11,50	102	32	$2 \cdot 32 + 256 =$ 320

4.2 Timings

In Table 2 we give timings for the two parameter sets. For comparability, we also give timings for the same operations on a personal computer (PC). The computer is an Intel Core Duo T7300 2GHz running Linux with kernel version 2.6.24. The application uses the same source code as the smart card implementation, compiled with GCC-4.1.3, optimization level O2.

The column labeled “time” lists the overall timing including the data transmission to and from the smart card. In the rightmost column we provide the time that is used by the mere computation on the card, excluding the transmission times. The gross bit rate of the transmission is 9600 bit/s. Please note that the SLE76 hardware platform generally supports much faster transmission rates than this.

Table 2. Timings for encryption and decryption operation of the McEliece PKC on a personal computer and the SLE76 smart card

platform	**parameter set**	**operation**	**time**	**time without I/O**
PC	m=10, t=40	encryption	0.75 ms	-
PC	“	decryption	0.8 ms	-
SLE76	“	encryption	1.26s	0.97s
SLE76	“	decryption	0.98s	0.69s
PC	m=11, t=50	encryption	1.2ms	-
PC	“	decryption	1.6ms	-
SLE76	“	encryption	1.85s	1.39s
SLE76	“	decryption	1.52s	1.06s

Concerning the encryption operation, we must point out that the measurement results are of small practical relevance. This is due to the fact that we perform the encryption by using a public key stored in the NVM of the device. In real life applications, this key would have to be exchanged for every new communication partner. Considering the public key size, this would cause totally impractical transmission times.

4.3 Resource Demands

In Table 3 we give the resource demands, i.e. the RAM and non-volatile memory (NVM) space needed by the implementation. Again, we distinguish the two parameter sets we are examining in this work. The demanded RAM size is made up of a fixed stack size of 1024 bytes and the peak amount of allocated heap memory. Note that the RAM demands given below are with respect to the decryption operation. They are lower for the encryption operation.

The main contribution to the private key size stems from the parity check matrix **H**, which makes up about 143,000 bytes in case of $m = 11$, $t = 50$ and about 53,000 bytes for $m = 10$, $t = 40$. This corresponds to portions of 95% and 88%, respectively. Please note that in addition to the raw matrix data, the given sizes also include certain management data overhead.

Table 3. Resource demands of the McEliece PKC smart card implementation with accuracy of 100 byte for RAM and 1000 byte for NVM

resource	**space in** 10^3 **byte**
RAM (m=11,t=50)	4.4
RAM (m=10,t=40)	3.4
NVM code	45
NVM public key (m=10,t=40)	33
NVM private key (m=10,t=40)	60
NVM public key (m=11,t=50)	106
NVM private key (m=11,t=50)	151
NVM $\mathbb{F}_{2^{10}}$ lookup tables	4
NVM $\mathbb{F}_{2^{11}}$ lookup tables	8

4.4 Key Generation

So far, our smart card implementation does not feature key generation. The reason for this is that this operation involves operations on matrices that by far exceed the card's RAM size. Since writing to the NVM takes much more time than writing to RAM, an optimized key generation algorithm would be needed in order to minimize those NVM write accesses.

In our implementation, we have realized a set of commands to write the private key parts from the PC to the card.

5 Conclusion and Outlook

The McEliece PKC, though existing for 30 years, has not experienced any serious use in real world applications so far. The main drawbacks of this scheme are the large sizes of the private and public key. But as shown by our work, the NVM and RAM provided by today's smart cards are already sufficient to support implementations of McEliece using parameters that provide about 100 bit security. Also, the achievable performance seems sufficient for certain applications.

The implementation presented in this work is fully functional, yet there are a number of possible improvements that could be applied to it. First of all, the code size could probably be reduced further by removing certain redundancies. Concerning the performance, a major improvement should result from the replacement of 32 bit pointers used throughout the code by 16 bit pointers. This is because the 16 bit CPU can handle the smaller pointers much faster. But since at least for the larger parameter set the private key size exceeds the 16 bit addressable area, this could only be achieved with the usage of the Memory Management Unit (MMU) available on the SLE76 platform. An alternative would be to use a 32 bit platform, of course.

Furthermore, our implementation will undergo a thorough analysis with respect to side channels and appropriate countermeasures will be incorporated.

The problem of the transmission and storage of the public key for the encryption operation, which arises in certain applications will also be addressed in future work.

Considering the fact that the McEliece PKC is providing security even in the presence of quantum computers, our results should encourage decision makers to examine applications of public key encryption schemes within their authority with respect to the need and possibility to switch to a quantum computer resistant scheme. As stated in the introduction, the replacement of the classical encryption schemes like RSA and elliptic curve based cryptography may not be delayed until the very moment at which potent quantum computers become available to attackers.

References

1. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976)
2. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21(2), 120–126 (1978)
3. Miller, V.: Use of Elliptic Curves in Cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
4. ElGamal, T.: A Public Key Cryptosystem and A Signature Based on Discrete Logarithms. IEEE Transactions on Information Theory (1985)
5. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings, 35th Annual Symposium on Foundation of Computer Science (1994)
6. Shor, P.W.: Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing 26(5), 1484–1509 (1997)
7. Proos, J., Zalka, C.: Shor's discrete logarithm quantum algorithm for elliptic curves. Technical Report quant-ph/0301141, arXiv (2006)
8. Merkle, R.: A Certified Digital Signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, Heidelberg (1990)
9. Buchmann, J., Garcia, L., Dahmen, E., Doering, M., Klintsevich, E.: CMSS-An Improved Merkle Signature Scheme. In: 7th International Conference on Cryptology in India-Indocrypt, vol. 6, pp. 349–363 (2006)
10. McEliece, R.J.: A public key cryptosystem based on algebraic coding theory. DSN progress report 42–44, 114–116 (1978)
11. Kobara, K., Imai, H.: Semantically secure McEliece public-key cryptosystems - conversions for McEliece PKC. In: Practice and Theory in Public Key Cryptography - PKC '01 Proceedings (2001)
12. MacWilliams, F.J., Sloane, N.J.A.: The theory of error correcting codes. North-Holland, Amsterdam (1997)
13. Goppa, V.D.: A new class of linear correcting codes. Problems of Information Transmission 6, 207–212 (1970)
14. Menezes, A., van Oorschot, P., Vanstone, S.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996), `http://www.cacr.math.uwaterloo.ca/hac/`
15. Engelbert, D., Overbeck, R., Schmidt, A.: A Summary of McEliece-Type Cryptosystems and their Security. Journal of Mathematical Cryptology (2007)

16. Canteaut, A., Chabaud, F.: A new algorithm for finding minimum-weight words in a linear code: application to primitive narrow-sense BCH-codes of length 511. IEEE Transactions on Information Theory 44(1), 367–378 (1998)
17. Bernstein, D.J., Lange, T., Peters, C.: Attacking and defending the McEliece cryptosystem. In: Buchmann, J., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 31–46. Springer, Heidelberg (2008)
18. Patterson, N.: Algebraic decoding of Goppa codes. IEEE Trans. Info. Theory 21, 203–207 (1975)
19. Overbeck, R.: An Analysis of Side Channels in the McEliece PKC (2008), `https://www.cosic.esat.kuleuven.be/nato_arw/slides_participants/Overbeck_slides_nato08.pdf`
20. Infineon Technologies AG: SLE76 Product Data Sheet, `http://www.infineon.com/cms/de/product/channel.html?channel=db3a3043156fd57301161520ab8b1c4c`
21. Biswas, B., Sendrier, N.: HyMES - Hybrid McEliece System, `http://ralyx.inria.fr/2008/Raweb/secret/uid18.html`
22. Biswas, B., Sendrier, N.: McEliece cryptosystem in real life: theory and practice. In: Buchmann, J., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 47–62. Springer, Heidelberg (2008)
23. Döring, M.: On the Theory and Practice of Quantum-Immune Cryptography. PHD-Thesis (2008), `http://www.cdc.informatik.tu-darmstadt.de/reports/README.diss.html`
24. The FlexiProvider group at Technische Universität Darmstadt: FlexiProvider, an open source Java Cryptographic Service Provider, `http://www.flexiprovider.de`

A Parity Check Matrix and Generator of an Irreducible Binary Goppa Code

The parity check matrix **H** of a Goppa code determined by the Goppa polynomial g can be determined as follows. $\mathbf{H} = \mathbf{XYZ}$, where

$$\mathbf{X} = \begin{bmatrix} g_t & 0 & 0 & \cdots & 0 \\ g_{t-1} & g_t & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & g_3 & \cdots & g_t \end{bmatrix}, \mathbf{Y} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \gamma_0 & \gamma_1 & \cdots & \gamma_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_0^{t-1} & \gamma_1^{t-1} & \cdots & \gamma_{n-1}^{t-1} \end{bmatrix},$$

$$\mathbf{Z} = \operatorname{diag}\left(\frac{1}{g(\gamma_0)}, \frac{1}{g(\gamma_1)}, \ldots, \frac{1}{g(\gamma_{n-1})}\right).$$

Here diag$\,(\ldots)$ denotes the diagonal matrix with entries specified in the argument. **H** is $t \times n$ matrix with entries in the field $\mathbb{F}_{2^m}$

As for any error correcting code, the parity check matrix allows for the computation of the syndrome of a distorted code word:

$$S_{\boldsymbol{z}}(X) = \boldsymbol{z}\mathbf{H}^\top \left(X^{t-1}, \cdots, X, 1\right)^\top.$$

The multiplication with $\left(X^{t-1}, \cdots, X, 1\right)^\top$ is used to turn the coefficient vector into a polynomial in $\mathbb{F}_{2^{mt}}$.

The generator of the code is constructed from the parity check matrix in the following way:

Transform the $t \times n$ matrix $\mathbf{H}$ over $\mathbb{F}_{2^m}$ into an $mt \times n$ matrix $\mathbf{H}_2$ over $\mathbb{F}_2$ by expanding the rows. Then, find an invertible matrix $\mathbf{S}$ such that

$$\mathbf{S} \cdot \mathbf{H}_2 = \left[\mathbb{I}_{mt} \mid \mathbf{R}^\top\right],$$

i.e., bring H into a systematic form using the Gauss algorithm. Here, $\mathbb{I}_x$ is the $x \times x$ identity matrix. Now take $\mathbf{G} = [\mathbb{I}_k \mid \mathbf{R}]$ as the public key. $\mathbf{G}$ is a $k \times n$ matrix over $\mathbb{F}_2$, where $k = n - mt$.

B The Extended Euclidean Algorithm (XGCD)

The extended Euclidean algorithm can be used to compute the greatest common divisor (gcd) of two polynomials[12].

In order to compute the gcd of two polynomials $r_{-1}(X)$ and $r_0(X)$ with $\deg(r_0)(X) \leq \deg(r_{-1}(X))$, we make repeated divisions to find the following sequence of equations:

$$\begin{array}{ll}
r_{-1}(X) = q_1(X)r_0(X) + r_1(X), & \deg(r_1) < \deg(r_0), \\
r_0(X) = q_2(X)r_1(X) + r_2(X), & \deg(r_2) < \deg(r_1), \\
\dots & \\
r_{i-2}(X) = q_i(X)r_{i-1}(X) + r_j(X), & \deg(r_i) < \deg(r_{i-1}), \\
r_{i-1}(X) = q_{i+1}(X)r_i(X) &
\end{array}$$

Then $r_i(X)$ is the gcd of $r_{-1}(X)$ and $r_0(X)$.