Journal of ELECTRICAL ENGINEERING, VOL. 65, NO. 6, 2014, 342–348

McELIECE PKC CALCULATOR

Marek Repka *

The original McEliece PKC proposal is interesting thanks to its resistance against all known attacks, even using quantum cryptanalysis, in an IND-CCA2 secure conversion. Here we present a generic implementation of the original McEliece PKC proposal, which provides test vectors (for all important intermediate results), and also in which a measurement tool for side-channel analysis is employed. To our best knowledge, this is the first such an implementation. This Calculator is valuable in implementation optimization, in further McEliece/Niederreiter like PKCs properties investigations, and also in teaching. Thanks to that, one can, for example, examine side-channel vulnerability of a certain implementation, or one can find out and test particular parameters of the cryptosystem in order to make them appropriate for an efficient hardware implementation. This implementation is available [1] in executable binary format, and as a static C++ library, as well as in form of source codes, for Linux and Windows operating systems.

K e y w o r d s: post-quantum PKI, McEliece PKC, Niederreiter PKC, Patterson's algebraic decoding algorithm, binary irreducible Goppa codes, side-channel analysis

1 INTRODUCTION

A symmetric cryptography is contemporary based on the integer factorization problem, or the discrete logarithm problem (very often defined over elliptic curves). An alternative to the common asymmetric cryptography is needed as the alternative should be based on a different problem that the common asymmetry, since by breaking those problems, the alternative cryptosystem will be untouched. The common asymetric cryptography is very vulnerable to the Shor's quantum algorithm [2], and therefore, the different problem must be also resistant against quantum cryptanalysis.

Fortunately, there is the McEliece PKC [3] (and also the derived Niederreiter PKC in a particular setup) that might prove resistant to attacks. The McEliece PKC is based on different problems like the usual ones. Those problems are also not affected by the Shor's algorithm [2]. It is based on the Coset Weights Decision NP-complete Problem, and the Subspace Weights Decision NP-complete Problem [4]. These problemes are from coding theory, and, thus, this is a code-based cryptograhy. For a very fresh survey of cryptography based on error correcting codes consult [5]. Using the McEliece PKC, or others code-based cryptosystems, one can costruct all the cryptographic primitives such as encryption, signature, hash functions, pseudo-random generators and so forth.

The original McEliece proposal, which uses random instances of binary irreducible Goppa code with the maximal length, has been unbroken since 1978, even considering quantum cryptanalysis (note the IND-CCA2 conversion importance [6]). There were many attempts to replace the underlying linear error correcting code but all attempts failed except the new proposal to use the Moderate Density Parity-Check Codes (MDPC) [7]. Since the MDPC proposal is very new, no one of the modifications of the original McEliece proposal is confident in the post-quantum sense today. Nowadays, there are only few, and very limited, implementations of the original proposal. Note also, that many derivatives of the original McEliece PKC have been published. For instance, the Niederreiter PKC [8] is equivalent to the McEliece PKC in terms of security but only if the Binary Irreducible Goppa codes with the maximal length are used. Other derivation is the HyMES [9], but the HyMES differs from the original McEliece PKC proposal in the key-pair and error vector generation, and thus, also in encryption as well as in decryption. A good implementaion of the HyMES with some countermeasures implemented and some improvements is the FLEA [10]. Further, a scheme that uses MDPC can be found in [11].

Since the derived schemes are young and, we can say, not sufficiently (quantum) cryptanalysed, no one of the derived schemes, except for the Niederreiter PKC in the proper setup, is confident as a post-quantum cryptosystem. This is the reason why we decided to implement the original McEliece PKC proposal.

We implemented the most generic original McEliece PKC proposal in order to make the PKC more available. The adjective generic has been achieved using the Number Theory Library (NTL) [12], and the generic CPU Tick Measurement Library [13]. Our implementation is called the McEliece PKC Calculator, since no parameter is fixed in this implementation, and test vectors for all the important intermediate results (for all appropriate m and t in limits of hardware and NTL) can be provided for any: encryption, decryption, or key generation. Thanks to the NTL, the Calculator is easy to understand, use, and modify, since the standard NTL functions, input, and output, are used. Therefore, if a key-pair not generated by the Calculator is desired to

* Institute of Computer Science and Mathematics, Faculty of Electrical Engineering and Information Technology, Ilkovičova 3, Bratislava, SK-812 19, Slovak Republic. marek.repka@stuba.sk

DOI: 10.2478/jee-2014-0056, Print ISSN 1335-3632, On-line ISSN 1339-309X

be used by Calculator, it is not a problem, it must be just formated accordingly. Moreover, a measurement tool for side-channel analysis has been employed, which test vectors can be also recorded for. Using this tool, timing leakage can be measured, and using the measured data, it is also possible to simulate power-consumption and electromagnetic-emanation leakages. The CPU Tick Library allows to measure CPU Ticks on different families of processors, and operation systems. To our best knowledge, this is the first such an implementation. Although, there exist several implementations, like [10, 14–17], of the original, and derived, schemes on hardware, embedded, and also on a computer platform. However, parameters are fixed, no test vectors are provided, or no tool for the side channel analysis is employed. The Calculator can be used in the PKC implementation optimization, and further McEliece/Niederreiter like PKCs properties investigation, as well as in proper key-pairs generation. The Calculator can be also used in proper parameter choice for a hardware implementation, and the leakage-measurement tool can provide information on side-channel vulnerabilities.

Since we have Post-Quantum PKC in the secure cryptosystem property setup [18], the only possibility how to break this PQ-PKC is via side-channel attacks. Recently, several side-channel attacks have been published [10, 19, 20, 21, 22]. It is possible to attack key generator, decryptor, and also encryptor. We stressed only the Patterson's algebraic decoding algorithm used in the decryption process. By the tool, secret error vector, secret permutation, and secret Goppa polynomial can be guessed, and the success rate of the guessing can be evaluated.

2 BINARY IRREDUCIBLE GOPPA CODES FOR THE McELIECE PKC

Goppa codes was invited by Goppa [23]. In the original McEliece PKC proposal, random instances of a binary irreducible Goppa code with maximal length are employed. These codes are proposed to be corrected by the Patterson's algebraic decoding algorithm, Fig. 5.

Let $\mathbb{F}_{2^m} = \mathbb{F}_2[X]/m(X)$ be the finite field, where $m(X)$ is an irreducible polynomial over $\mathbb{F}_2[X]$, and $\deg m(X) = m$.

2.1 Binary Irreducible Goppa Polynomial Definition

Binary Irreducible Goppa polynomial is a monic binary irreducible polynomial $g(Z) \in \mathbb{F}_{2^m}[Z]$, where $\deg g(Z) = t$.

2.2 Code Support Definition

Code support is a vector $\mathbf{\Lambda} \in \mathbb{F}_{2^m}^n$, $\mathbf{\Lambda} = (\lambda_i)_{0 \le i \le n-1}$ consisting of pairwise distinct elements $\lambda_i \in \mathbb{F}_{2^m}$, where $g(\lambda_i) \neq 0$.

Since the Goppa polynomial $g(Z)$ is irreducible, all the field elements are in the code support. Hence, the code length $n = 2^m$.

2.3 The Code Definition

Binary Irreducible Goppa code $\Gamma(\mathbf{\Lambda}, g)$ is a Linear Alternant code defined over $\mathbb{F}_{2^m}$, wherein the g is a binary irreducible Goppa polynomial, and the $\mathbf{\Lambda}$ is a code support. This code has parameters $[n = 2^m, k = n - mt, d = 2t + 1]$, and it is defined as follows:

$$\Gamma(\mathbf{\Lambda}, g) := \{\boldsymbol{c} \in \mathbb{F}_2^n \colon S(\boldsymbol{c}, Z) \equiv 0 \bmod g(Z)\}\ , \quad (1)$$

where

$$S(\boldsymbol{c}, Z) = \sum_{0 \le i \le n-1} \frac{c_i}{Z - \lambda_i} \quad (2)$$

is its syndrome polynomial.

Note, if we have $\boldsymbol{c} \in \Gamma(\mathbf{\Lambda}, g)$, $\boldsymbol{y} \in \mathbb{F}_2^n$, and $\boldsymbol{y} = \boldsymbol{c} \oplus \boldsymbol{e}$, where $\boldsymbol{e}$ is an error vector of $0 \le \mathrm{HW}(\boldsymbol{e}) \le t$, then $S(\boldsymbol{e}, Z) \equiv S(\boldsymbol{c}, Z) \bmod g(Z)$.

2.4 Error-Locator Polynomial Definition

The error-locator polynomial $\sigma(\boldsymbol{e}, Z)$ is defined, in binary case, as follows:

$$\sigma(\boldsymbol{e}, Z) = \prod_{i=0}^{n-1} (Z - \lambda_i)^{e_i}\ . \quad (3)$$

The polynomial is defined over $\mathbb{F}_{2^m}[Z]$, and indexes of its roots in the code support determine error-bit positions in a codeword. In our case, roots are not multiple, and its degree is t.

3 CALCULATOR IMPLEMENTATION DETAILS

As we stated in the Introduction, Calculator implementation is based on the Number Theory Library (NTL), and the generic CPU Tick Measurement Library. This provides the Calculator by the features we also stated in the Introduction. What is important to note here is that whenever possible, the NTL methods are used. For instance, the NTL is used for random polynomial generation, addition and multiplication, exponentiation, polynomial evaluation, inversion, extended euclidean algorithm, gaussian elimination, factorization, random error and messages vectors generation. Description of the implementation details follows.

Require: $m(X)$, code length n, and $t = \deg g(Z)$.
Ensure: $K_{pub} = G_{pub}$, $K_{priv} = \big(\Gamma(\mathbf{\Lambda}, g), S, P\big)$.
1. Generate uniformly a random $g(Z)$.
   ▷ Determines the secret $\Gamma(\mathbf{\Lambda}, g)$.
2. Find a generator matrix G_{priv} for the random secret $\Gamma(\mathbf{\Lambda}, g)$ code. ▷ Eq. (6), (7), (8)
3. Generate uniformly a random $k \times k$ dense invertible binary matrix S.
4. Generate uniformly a random $n \times n$ binary permutation matrix P. ▷ Alg. in Fig. 2
5. $G_{pub} = SG_{priv}P$.
6. $K_{pub} = G_{pub}$.
7. $K_{priv} = \big(\Gamma(\mathbf{\Lambda}, g), S, P\big)$.
8. **return** K_{pub}, and K_{priv}.

Fig. 1. McEliece PKC key generation

Require: $\boldsymbol{p}$, a sequence of elements for permuting.
Ensure: $\boldsymbol{p}$ with randomly permuted elements.
1. **for** $i = 0$; $i < \boldsymbol{p}$.length; i++ **do**
2. index = rand() mod $\boldsymbol{p}$.length
3. swap($\boldsymbol{p}$[index], $\boldsymbol{p}$[i])
4. **end for**
5. **return** $\boldsymbol{p}$

Fig. 2. Random permutation of a sequence of elements

3.1 Key-Pairs Generation

Private key K_{priv} consist of a random $\Gamma(\mathbf{\Lambda}, g)$ code, a random permutation matrix P, and a random dense non-singular scramble matrix S.

$$K_{priv} = (\Gamma(\mathbf{\Lambda}, g), S, P)\ . \tag{4}$$

The random $\Gamma(\mathbf{\Lambda}, g)$ code means that the g is chosen randomly. The public key K_{pub} is derived from the K_{priv} using P and S. The key generation algorithm is in Fig. 1.

$$K_{pub} = G_{pub}\,. \tag{5}$$

As the first step in the key generation phase, the Calculator picks up randomly (or it is chosen by an user) an irreducible polynomial $m(X)$ over $\mathbb{F}_2[X]$, according to that the finite field $\mathbb{F}_{2^m}$ is created. Then a binary irreducible Goppa polynomial $g(Z)$ over $\mathbb{F}_{2^m}[Z]$ is generated randomly. Probability that a random polynomial with degree t is irreducible over the $\mathbb{F}_{2^m}[Z]$ is approximately $1/t$ [3].

Now, the code support is initialized. All the elements of $\mathbb{F}_{2^m}$ are in the support. If the $m(X)$ polynomial is primitive, all elements can be generated using its roots. But it is not the case in general. Therefore, in order to initialize the code support, a generator (field primitive element) should be found. The Calculator searches for a generator using the fact that order of a subgroup divides order of the group. Order of an element that generates the field should be $n-1$. Let we have all the factors of the integer $n-1$. The generator is found by examination degrees of all the field elements respectively. If an element is found that has the desired degree, the search stops, and the element is used to initialize the code support. We denote this element as λ_1. The first element λ_0 of the initialized code support is always 0.

Generator matrix G_{priv} is found as follows. First, an initial parity check matrix H_{init} is constructed as

$$H_{init}(i,j) = g^{-1}(\lambda_j)\lambda_j^i\,, \tag{6}$$

where $H_{init}(i,j)$ is the i-th row, and the j-th column of the H_{init}. This matrix is then used in its binary form. Therefore, each cell (element of the finite filed $\mathbb{F}_{2^m}$) is represented as the column of sequence m binary digits. Thus, the matrix in the binary form consists of mt rows and n columns. Only the binary form of the parity check matrix is considered hereafter.

The parity check matrix is brought into the reduced row-echelon form using the Gaussian elimination. If the resulted matrix is not in the systematic form, the systematic form is obtained by swapping appropriate columns. Now we have

$$H_{init} = [I|R]\,. \tag{7}$$

The parity coordinates generator matrix $R^\top$ has $(n - mt)$ rows and mt columns. Using $R^\top$, the G_{priv} is then defined as

$$G_{priv} := [R^\top|I]\,. \tag{8}$$

In order to be able to construct the secret code support $\mathbf{\Lambda}$ for the code generated by the secret G_{priv}, the permutation of elements of the initial code support must be corrected according the swaps performed in order to make (7) held. For that purpose, only vector of the inverse swaps is important. The vector will be denoted as $\boldsymbol{b}$ hereafter. Note, the initial parity check matrix H_{init} is not a parity check matrix for the $\Gamma(\mathbf{\Lambda}, g)$ code generated by G_{priv}, since the code support correction.

The dense non-singular matrix S is generated randomly and uniformly by NTL. A random square matrix is invertible with probability approximately $1/3$. One possibility how to determine whether a square matrix is invertible is to examine its determinant but this would be time consuming. A better approach is to test the actual diagonal element for zero during the elimination when bringing the square matrix into an upper echelon form. If the element is zero the matrix is not invertible. For further optimization, inner instruction parallelism can be used [24].

The permutation matrix P is generated randomly and uniformly using the algorithm shown in Fig. 2. This algorithm assume a vector $\boldsymbol{p}$ which is somehow initialized. Hereafter, we consider $\boldsymbol{p}$ as vector of randomly and without replacement generated integers (a random permutation) that represents the secret permutation matrix P.

Require: $K_{pub} = G_{pub}$, end message $\boldsymbol{a} \in \mathbb{F}_2^k$, where $k = 2^m - mt$ (the number of rows of G_{pub}).
Ensure: A ciphertext $\boldsymbol{y} \in \mathbb{F}_2^n$. $\triangleright\ n = 2^m$.
1. Generate uniformly a random binary vector $\boldsymbol{e}_x \in \mathbb{F}_2^n$ with $\mathrm{HW}(\boldsymbol{e}_x) = t$. $\triangleright$ HW is Hamming weight.
2. $\boldsymbol{x} = \boldsymbol{a}G_{pub}$.
3. $\boldsymbol{y} = \boldsymbol{x} \oplus \boldsymbol{e}_{\boldsymbol{x}}$.
4. **return** y.

Fig. 3. McEliece PKC encryption

Require: $K_{priv} = \big(\Gamma(\boldsymbol{\Lambda}, g), S, P\big)$, and a ciphertext $\boldsymbol{y} \in \mathbb{F}_2^n$.
Ensure: Message $\boldsymbol{a} \in \mathbb{F}_2^k$.
1. $\boldsymbol{u} = \boldsymbol{y}P^{-1}$. $\triangleright\ \boldsymbol{u} = \boldsymbol{a}SG_{priv} + \boldsymbol{e}_{\boldsymbol{x}}P^{-1}$, the vector $\boldsymbol{p}$ is used instead of P.
2. $\boldsymbol{e} = \texttt{Patterson}\big(\boldsymbol{u}, \Gamma(\boldsymbol{\Lambda}, g)\big)$. $\triangleright\ \boldsymbol{e} = \boldsymbol{e}_{\boldsymbol{x}}P^{-1}$, Fig. 5.
3. $\boldsymbol{v} = \boldsymbol{u} + \boldsymbol{e}$. $\triangleright\ \boldsymbol{v} = \boldsymbol{a}SG_{priv}$.
4. $\boldsymbol{w} = \texttt{GetInformationCoordinates}(\boldsymbol{v})$. $\triangleright\ \boldsymbol{w} = \boldsymbol{a}S$, the last $n-k$ coordinates of $\boldsymbol{v}$.
5. $\boldsymbol{a} = \boldsymbol{w}S^{-1}$.
6. **return** $\boldsymbol{a}$.

Fig. 4. McEliece PKC decryption

Require: $\boldsymbol{u} \in \mathbb{F}_2^n$ (a private code word with t errors), $\Gamma(\boldsymbol{\Lambda}, g)$.
Ensure: Error vector $\boldsymbol{e}$ such that $\boldsymbol{v} = \boldsymbol{u} + \boldsymbol{e}$, where $\boldsymbol{v} \in \Gamma(\boldsymbol{\Lambda}, g)$ is the code word.
1. $S(\boldsymbol{e}, Z) \equiv S(\boldsymbol{u}, Z) \bmod g(Z)$. $\triangleright$ Eq. 9
2. $T(\boldsymbol{e}, Z) \equiv S^{-1}(\boldsymbol{e}, Z) + Z \bmod g(Z)$. $\triangleright$ EEA
3. $\tau(\boldsymbol{e}, Z) \equiv \sqrt{T(\boldsymbol{e}, Z)} \bmod g(Z)$. $\triangleright$ Eq. 13
4. Find $\alpha(\boldsymbol{e}, Z)$ and $\beta(\boldsymbol{e}, Z)$ such that $\beta(\boldsymbol{e}, Z)\tau(\boldsymbol{e}, Z) \equiv \alpha(\boldsymbol{e}, Z) \bmod g(Z)$. $\triangleright$ EEA
5. $\sigma(\boldsymbol{e}, Z) = \alpha^2(\boldsymbol{e}, Z) + Z\beta^2(\boldsymbol{e}, Z)$. $\triangleright$ Squaring
6. Find roots of $\sigma(\boldsymbol{e}, Z)$. $\triangleright$ Evaluation over the $\boldsymbol{\Lambda}$
7. Determine indexes of the roots in the support $\boldsymbol{\Lambda}$.
8. Set 1 in the determined indexes in error vector $\boldsymbol{e}$.
9. **return** $\boldsymbol{e}$.

Fig. 5. Patterson's algebraic decoding algorithm.

3.2 Key-Pairs Storing

In the Calculator implementation, almost nothing is fixed, even polynomial $m(X)$ is chosen randomly, or can be chosen by user. For the private key K_{priv} reconstruction: the $m(X)$, finite field generator element λ_1, vector of inverse swaps $\boldsymbol{b}$, permutation vector $\boldsymbol{p}$, matrix S, and, finally, $g(Z)$, are stored. In case of $m = 11$, $t = 50$, it is 4 510 452 bytes.

In order to reconstruct the corresponding public key $K_{pub} = G_{pub}$, only the G_{pub} is stored. For $m = 11$, $t = 50$, it is 6 138 820 bytes. We recommend to use any compression method in order to safe the size needed for key-pairs storing. Another possibility is to order permutations in the way that each permutation can be represented by an unique integer [25].

3.3 Encryption

The encryption algorithm is very fast and simple, Fig. 3. It can be implemented as several XOR additions in an optimized implementation. In order to generate uniformly a random secret error vector of hamming weight t and length n, the Calculator implementation uses the algorithm listed in Fig. 2. From the outcome of the algorithm, only the last t indexes are considered. These indexes determines positions of ones in the error vector.

3.4 Decryption

The decryption algorithm is more time consuming than the encryption one. The most time consuming is the Step 2.

The Patterson algebraic decoding algorithm (Fig. 5) is used in order to correct a code word with t errors in the private $\Gamma(\boldsymbol{\Lambda}, g)$ code. For that purpose, we assume an input binary vector $\boldsymbol{u} = \boldsymbol{y}P^{-1}$ that is a codeword in the private code with exactly t errors. Note, t is the maximum number of errors that can be corrected in a $\Gamma(\boldsymbol{\Lambda}, g)$ code, and also that the algorithm in Fig. 5 is capable to correct.

As the first step of the Fig. 5, the syndrome of the error vector is computed. One can compute the syndrome evaluating the syndrome polynomial (2), but such an evaluation would be the most time consuming step in the decoding algorithm. On the other hand, such an evaluation is very useful on a memory constraint devices. In order to speed up the syndrome evaluation, following look-up table is precomputed $\forall\, 0 \leq i < n$

$$\texttt{preSynTab}[i] = (Z - \lambda_i)^{-1} \bmod g(Z)\,. \tag{9}$$

Next possibility how to compute the syndrome is to compute the product $uH_{priv}^{\top}$, wherein the H_{priv} is a parity-check matrix of the secret $\Gamma(\boldsymbol{\Lambda}, g)$ code, and obtain the syndrome in this way.

The syndrome polynomial $S(\boldsymbol{e}, Z)$ satisifies

$$S(\boldsymbol{e}, Z) \equiv \frac{\sigma'(\boldsymbol{e}, Z)}{\sigma(\boldsymbol{e}, Z)} \bmod g(Z)\,. \tag{10}$$

Since the error of a word is being determined, the first derivative of the error-locator polynomial consists only of all the even terms, *ie* the error-locator polynomial can be split into squares and non-squares

$$\sigma(\boldsymbol{e}, Z) = \alpha^2(\boldsymbol{e}, Z) + Z\beta^2(\boldsymbol{e}, Z)\,, \tag{11}$$

where $\beta^2(\boldsymbol{e}, Z) = \sigma'(\boldsymbol{e}, Z)$. After few modifications, the Key Equation can be obtained as

$$\beta(\boldsymbol{e}, Z)\sqrt{S^{-1}(\boldsymbol{e}, Z) + Z} \equiv \alpha(\boldsymbol{e}, Z) \bmod g(Z)\,. \tag{12}$$

Therefore, in the Step 3 of Fig. 5, the square-root modulo $g(Z)$ is computed. Let us denote the term $S^{-1}(\boldsymbol{e}, Z) + Z$ as $T(\boldsymbol{e}, Z)$. The Calculator implementation uses the fact of the perfect square, and thus

$$\tau(\boldsymbol{e}, Z) = \sqrt{T(\boldsymbol{e}, Z)} \equiv T^{2^{tm-1}}(\boldsymbol{e}, Z) \bmod g(Z)\,. \tag{13}$$

Such an approach can be very useful on memory constraint devices, but on the other hand it is very time consuming operation. Another possibility how to compute that square-root is to use precomputed look-up table, which consist of $\tau_i(Z)$ such that $\tau_i^2(Z) \equiv Z^i \bmod g(Z)$ for $0 \leq i < t$.

Subsequently, the key equation is solved using the Extended Euclidean Algorithm (EEA) that stops when $\deg \alpha_j(Z) \leq \lfloor (t+1)/2 - 1 \rfloor \leq t/2$, where j is the EEA iteration number.

At the time the error-locator polynomial $\sigma(\boldsymbol{e}, Z)$ is computed, roots of the error-locator polynomial shall be found. The Calculator implementation simply evaluate the $\sigma(\boldsymbol{e}, Z)$ over the secret code support $\boldsymbol{\Lambda}$. Therefore, Steps 6, 7, 8 are conducted in one loop. This method is also time consuming, and, as an alternative, any other factorization method can be employed. Thus, the decoding algorithm yields the error vector $\boldsymbol{e}$ in the private code.

When the secret error vector $\boldsymbol{e}$ is removed Step 3 in Fig. 4, only information coordinates are addressed, and the scrambling matrix is removed. Finally, the decrypted message is obtained.

4 BASIC USE CASES

Next we list basic use cases in order to provide examples of Calculator usage.

4.1 Regular Cryptosystem

The Calculator can be used as a basic cryptosystem, for key-pairs generation, encryption, as well as for decryption. It is very important to note that the original McEliece PKC is vulnerable to (adaptive) chosen-ciphertext attacks. Therefore, the Calculator can be used for encryption and decryption only if it is plugged into an IND-CCA2-Secure conversion, like the γ conversion defined in [26].

For key-pairs generation, encryption, and decryption, the following commands can be used respectively:

```
keygen m t privateKeyFileName publicKeyFilename
enc publicKeyFilename inFileName outFileName
dec privateKeyFilename inFileName outFileName
```

4.2 PKC Implementation Optimization, and Properties Investigation

Essentially, the first main purpose of the Calculator development was the PKC implementation optimization for an FPGA. Test vectors have been used in order to chose particular PKC parameters that we have fixed for the implementation in FPGA. Afterwards, test vectors have been used for the FPGA implementation validation.

Further, test vectors can be used in the further McEliece like PKCs properties investigation because all important intermediate results are recorded. The intermediated results can be used in order to verify stated hypotheses, or jut to trace behavior. For the PKC properties investigation, also the information recorded by the side-channel-leakage measurement tool can be used.

The test vectors recording can be turned-on appending any command by a file name for the test vectors file. Test vectors are formatted using the standard NTL output.

4.3 Side-Channel Vulnerability Examination

The side-channel-leakage measurement tool records Indicators, Tab. 2, measured in order to preform an attack, and information about secret, Tabs. 3 and 4, used to compute success rate of an attack. Secret error vector, secret permutation, and secret Goppa polynomial, respectively can be guessed using the measured data. Also power-consumption and electromagnetic-emanation leakages can be simulated. Using this tool, particular keys, and proposed countermeasures can be tested. Thanks to the NTL, source codes are easy to read, and it is possible to replace a measured operation for a designer's one. Not only computation time is measured by the tool. Also Degrees and Hamming Weights of polynomials processed are recored.

```
measure--key measurementCode publicKeyFilename
        privateKeyFilename nTests min_hw_e
        max_hw_e measurementFileName
```

```
measure--rnd--keys measurementCode m t nRandKeyPairs
       nTests publicKeyFilename min_hw_e max_hw_e
       measurementFileName is_storeKeys
```

5 THE SIDE–CHANNEL–LEAKAGE MEASUREMENT TOOL

This measurement tool is employed in the Patterson's algebraic decoding algorithm, Fig. 5. As we mentioned above, using the tool, secret error vector, secret permutation, and also the secret Goppa polynomial, can be possible to guess. Moreover, power consumption and electromagnetic emanation leakages can be simulated using the measured data provided by this tool.

5.1 Measurement Type 1

This measurement type records average computation time, standard deviation of the computation time, and if applicable, average values and standard deviations of Hamming Weights and degrees of polynomials processed, and steps performed, during Patterson's algebraic decoding algorithm defined in Fig. 5. The purpose is to measure these Indicators (Tab. 2) dependency on $\mathrm{HW}(\boldsymbol{e})$, see Tab. 1. Hence, output file of this measurement type is composed as follows. As the first column there is the $\mathrm{HW}(\boldsymbol{e})$ growth according to that Indicators are recorded. Since there is a possibility to make such record for many

Table 1. Information recorded about secret error vector in order to measure success rate of an attack

Item/Column number	The secret value
1	HW($\boldsymbol{e}$)

Table 2. Indicators measured in order to be able to perform side-channel attacks, and determine where the leakages occur

Item/Column number	Fig. 5 step	Indicators: `avg`$(\cdot)$, `atd`$(\cdot)$	Object
1, ... ,6	1.	computation time deg, HW	$S(Z)$
7, ... ,12	2.	computation time deg, HW	$T(Z)$
13, ... ,18	3.	computation time deg, HW	$\tau(Z)$
19, 20	4.	computation time	EEA
21, ... ,24	4.	deg, HW	$\alpha(Z)$
25, ... ,30	5.	computation time deg, HW	$\sigma(Z)$
31, 32	6.,7.,8.	computation time	$\boldsymbol{e}$ construction

Table 3. One measurement file header for measurement setup.

Value	Description
m	$\mathbb{F}_{2^m}$
t	deg $g(Z)$
nRandKeyPairs	Number of randomly generated key-pairs
nTests	Number of random messages per key-pair and HW($\boldsymbol{e}$)
min(HW($\boldsymbol{e}$))	Start HW($\boldsymbol{e}$)
max(HW($\boldsymbol{e}$))	End HW($\boldsymbol{e}$)

Table 4. Information recorded about secret Goppa polynomial in order to measure success rate of an attack

Item/Column number	The secret value
$1, \ldots, (t+1)$	$g_0, \ldots, g_t$
$(t+2), \ldots, (2t+3)$	$\mathrm{HW}(g_0), \ldots, \mathrm{HW}(g_t)$
$(2t+4)$	$\mathrm{HW}(g(Z))$

random key-pairs, for the next key-pair there is the next such record separated by empty row. As the last data in the measurement file, summarization over all the key-pairs is placed. At the beginning of measurement file the measurement setup as stated in Tab. 3 is placed.

The average values and the standard deviations are computed from nTests encryptions. The measurement file contains nRandKeyPairs measurement records, each for one random private key. Optionally, also the test vectors can be stored in the disk as a text file.

5.2 Measurement Type 2

This second measurement type is designated to measure Indicators (Tab. 2) dependency on secret Goppa polynomial (Tab. 4). Corresponding to each information about Goppa polynomial, Indicators are measured. If moreover dependency on secret permutation is desired to measure, then the flag `is_storeKeys` must be set to 1. Output file of a measurement of this type is composed as follows.

At the beginning of the file, the measurement setup is presented. The measurement setup is arranged in Tab. 3. Regarding the measurement setup, afterwards, information about Goppa polynomials (Tab. 4), HW($\boldsymbol{e}$) (Tab. 1), and the measured Indicators (Tab. 2) are stored form left to right respectively. Thus, for each Goppa polynomial there is a row, which displays also step-by-step HW($\boldsymbol{e}$) and indicators for each $i \in [\min(\mathrm{HW}(\boldsymbol{e})), \max(\mathrm{HW}(\boldsymbol{e}))]$ according the measurement setup.

Such as in the measurement type 1, the average values and the standard deviations are computed from nTests encryptions. The measurement file contains nRandKeyPairs measurement rows, each for one random private key. Optionally, also the test vectors can be stored in the disk as a text file.

6 CONCLUSIONS

We implemented a Calculator that allows to choice all parameters for the original McEliece PKC. It can provide test vectors and information about a side-channel vulnerably. IND-CCA2 security is not addressed in this work, however the Calculator can be taken and the IND-CCA2 secure conversion can be made without any implementation modification. Although this PKC is believed to be unbreakable even using quantum cryptanalysis, it must also be implemented into a real device. Thus, post-quantum McEliece PKC, or Niederreiter PKC, are not an exception in terms of Side-Channel Attacks. Moreover, attacks employing SCAs considering the new method to solve Multiple right-hand side equation systems [27] (Algebraic SCAs) are indeed a serious threat in post-quantum cryptography. Designed countermeasures against SCA can be evaluated, when add to the code of Calculator.

Acknowledgment

Supported in part by NATOs Public Diplomacy Division in the framework of Science for Peace SPS Project 98452; the EC FP7-SEC-2011-285205 FREESIC project; grant APVV-0586-11; grant VEGA 1/0173/13; National Scholarship Programme of SR – SAIA, n. o.; Laboratoire Hubert Curien UMR CRNS 5516; and the Slovak TEMPEST, a. s. Company.

References

[1] S. P. N. 984520, Secure Implementation of Post-Quantum Cryptography, 2013. http://147.175.106.232/nato/?q=node/27.

[2] SHOR, P.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer, SIAM review **41** No. 2 (1999), 303–332.

[3] McELIECE, R. J.: A Public-Key Cryptosystem based on Algebraic Coding Theory, DSN Progress Report **42** No. 44 (1978), 114–116, http://www.cs.colorado.edu/~jrblack/class/csci7000/f03/papers/mceliece.pdf.

[4] BERLEKAMP, E.—McELIECE, R.—van TILBORG, H.: On the Inherent Intractability of Certain Coding Problems, IEEE Transactions on Information Theory **24** No. 3 (May 1978), 384–386.

[5] REPKA, M.—CAYREL, P.-L.: Multidisciplinary Perspectives in Cryptology and Information Security, IGI Global, Ch. Cryptography based on Error Correcting Codes: a Survey, 2014, pp. 133–156.

[6] ZAJAC, P.: A Note on CCA2-Protected McEliece Cryptosystem with a Systematic Public Key, Cryptology ePrint Archive, Report 2014/651, 2014 http://eprint.iacr.org/.

[7] OUZAN, S.—BE'ERY, Y.: Moderate-Density Parity-Check Codes, CoRR **abs/0911.3262** (2009).

[8] NIEDERREITER, H.: Knapsack-Type Cryptosystems and Algebraic Coding Theory, Problems of Control and Information Theory **15** No. 2 (1986), 159–166.

[9] BISWAS, B.—SENDRIER, N.: Mceliece Cryptosystem Implementation: Theory and Practice, in PQCrypto, ser. LNCS (J. Buchmann and J. Ding, eds.), vol. 5299, Springer, 2008, pp. 47–62.

[10] STRENZKE, F.: Efficiency and Implementation Security of Code-Based Cryptosystems, PhD dissertation, Universität Darmstadt, Germany, 2013.

[11] HEYSE, S.—von MAURICH, I.—GÜNEYSU, T.: Smaller Keys for Code-Based Cryptography: Qc-mdpc Mceliece Implementations on Embedded Devices, in CHES, ser. LNCS (G. Bertoni and J.-S. Coron, eds.), vol. 8086, Springer, 2013, pp. 273–292.

SHOUP, V.: Ntl: A Library for Doing Number Theory (version 6.0.0), February 2013. http://www.shoup.net/ntl/.

FRIGO, M.—JOHNSON, S. G.: The Design and Implementation of FFTW3, Proceedings of the IEEE **93** No. 2 (2005), 216–231, Special Issue on Program Generation, Optimization, and Platform Adaptation. http://www.fftw.org/.

[14] EISENBARTH, T.—GÜNEYSU, T.—HEYSE, S.—PAAR, C.: Microeliece: Mceliece for Embedded Devices, in CHES, ser. LNCS (C. Clavier and K. Gaj, eds.), vol. 5747, Springer, 2009, pp. 49–64.

STRENZKE, F.: A Smart Card Implementation of the mceliece pkc, in The 4th IFIP WG 11.2, Proceedings, ser. WISTP'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 47–59.

[16] CAYERL, P.-L.: Code Based Cryptography, 2012, http://cayrel.net/research/code-based-cryptography/code-based-cryptosystems/.

[17] HEYSE, S.—GÜNEYSU, T.: Towards One Cycle Per Bit Asymmetric Encryption: Code-Based Cryptography on Reconfigurable Hardware, in CHES, ser. LNCS (E. Prouff and P. Schaumont, eds.), vol. 7428, Springer, 2012, pp. 340–355.

[18] ENGELBERT, D.—OVERBECK, R.—SCHMIDT, A.: A Summary of Mceliece-Type Cryptosystems and their Security, J. Mathematical Cryptology **1** No. 2 (2007), 151–199.

[19] SENDRIER, N. Ed.: PQCrypto 2010, Darmstadt, Germany, May 25-28, 2010, Proceedings, ser. LNCS, vol. 6061, Springer, 2010.

[20] STRENZKE, F.: A Timing Attack against the Secret Permutation in the Mceliece pkc, in PQCrypto ser. LNCS (N. Sendrier, eds.), vol. 6061, Springer, 2010, pp. 95–107.

[21] STRENZKE, F.: Timing Attacks Against the Syndrome Inversion in Code-Based Cryptosystems, in PQCrypto, ser. LNCS (P. Gaborit, ed.), vol. 7932, Springer, 2013, pp. 217–230.

[22] STRENZKE, F.—TEWS, E.—MOLTER, H. G.—OVERBECK, R.—SHOUFAN, A.: Side Channels in the Mceliece pkc, in PQCrypto, ser. LNCS (J. Buchmann and J. Ding, eds.), vol. 5299, Springer, 2008, pp. 216–229.

[23] GOPPA, V. D.: A New Class of Linear Error Correcting Codes, Probl. Pered. Inform. **6** (Sep 1970), 24–30.

[24] ZAJAC, P.—JOKAY, M.: Computing Indexes and Periods of All Boolean Matrices up to Dimension $n = 8$, Computing and Informatics **31** No. 6 (2013), 1329–1344.

[25] SEDGEWICK, R.: Permutation Generation Methods, ACM Comput. Surv. **9** No. 2 (June 1977), 137–164, http://doi.acm.org/10.1145/356689.356692.

[26] KOBARA, K.—IMAI, H.: Semantically Secure Mceliece Public-Key Cryptosystems-Conversions for Mceliece pkc, in Public Key Cryptography, ser. LNCS (K. Kim, ed.), vol. 1992, Springer, 2001, pp. 19–35.

[27] ZAJAC, P.: A New Method to Solve mrhs Equation Systems and its Connection to Group Factorization, J. Mathematical Cryptology **7** No. 4 (2013), 367–381.

[28] PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings, ser. LNCS (J. Buchmann and J. Ding, eds.), vol. 5299, Springer, 2008.

Received 13 March 2014

Marek Repka is a PhD student at FEI STU since 2010 year. Applied informatics focused on security is aim of his study, within which he finished also his bachelor and master degrees. His supervisor is Prof. Otokar Grošek. Marek is information security professional focused mainly on Side Channel Analysis of cryptosystems, Application Security, and Implementation and Integration of Security Controls. He was born in Uherské Hradiště, Czech Republic, in 1985.