Side Channels in the McEliece PKC

Falko Strenzke[1], Erik Tews[2], H. Gregor Molter[3], Raphael Overbeck[4], and Abdulhadi Shoufan[3]

[1] FlexSecure GmbH, Germany*
strenzke@flexsecure.de
[2] Cryptography and Computeralgebra, Department of Computer Science, Technische Universität Darmstadt, Germany
e_tews@cdc.informatik.tu-darmstadt.de
[3] Integrated Circuits and Systems Lab, Department of Computer Science, Technische Universität Darmstadt, Germany
{molter,shoufan}@iss.tu-darmstadt.de
[4] Ecole Polytechnique Fédérale de Lausanne, Switzerland
raphael.overbeck@epfl.ch

Abstract. The McEliece public key cryptosystem (PKC) is regarded as secure in the presence of quantum computers because no efficient quantum algorithm is known for the underlying problems, which this cryptosystem is built upon. As we show in this paper, a straightforward implementation of this system may feature several side channels. Specifically, we present a Timing Attack which was executed successfully against a software implementation of the McEliece PKC. Furthermore, the critical system components for key generation and decryption are inspected to identify channels enabling power and cache attacks. Implementation aspects are proposed as countermeasures to face these attacks.

Keywords: side channel attack, timing attack, post quantum cryptography.

1 Introduction

Current cryptographic systems depend on complex mathematical problems such as the factorization of large prime numbers and the calculation of discrete logarithms [1,2,3,4]. These systems are known to be vulnerable against certain algorithms which could be implemented efficiently on quantum computers [5,6,7]. New classes of cryptographic schemes will be needed to guarantee system and network security also in the presence of quantum computers. Examples for theses classes are the hash-based cryptography, such as the Merkle signature scheme [8,9], and code-based cryptography such as McEliece PKC [10,11].

The McEliece PKC is based on Goppa codes. The strongest known attack is based on solving the NP-hard decoding problem, and no quantum algorithm has been proposed which increases the efficiency of this attack [12]. So, although

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

J. Buchmann and J. Ding (Eds.): PQCrypto 2008, LNCS 5299, pp. 216–229, 2008.
© Springer-Verlag Berlin Heidelberg 2008

well-studied regarding its security against algorithm attacks, to the best of our knowledge, the McEliece PKC has never been analyzed with respect to side channel attacks. Side channel attacks target a cryptographic system taking advantage of its implementation [13,14,15,16]. Algorithm execution is associated with measurable quantities such as power consumption and execution time. The amounts of these quantities depend on the data processed by the algorithm. If the processed data is secret such as a private key, then the measured quantities may disclose the secret totally or partially. To prevent side channel attacks, countermeasures must be included during the implementation of the algorithm.

Our contribution

This paper addresses side channel attacks on the McEliece PKC and corresponding countermeasures. It is constructed as follows. Section 2 presents as preliminaries the Goppa code and the McEliece PKC in brief. Section 3 details a timing attack on the degree of error locator polynomial, which is used in the error correction step in the decryption algorithm. A theoretical justification for this attack is presented as well as experimental results of the execution of the attack against a software implementation. Also, countermeasures are addressed. Section 4 outlines two other side channel attacks and related countermeasures: a power attack on the construction of the parity check matrix during key generation and a cache attack on the permutation of code words during decryption. Section 5 concludes the paper.

2 Preliminaries

In this section we assume that the reader is familiar with the basics of error correction codes. We use the notation given e.g. in [17].

2.1 Goppa Codes

Goppa codes [18] 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*

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

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

Then an irreducible binary Goppa code is defined as

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

where $n = 2^m$*,* $S_{\boldsymbol{c}}(Y)$ *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. The canonical check matrix H for $\mathcal{G}(\mathbb{F}_{2^m}, g(Y))$ 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. 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 [19].

Here, we describe the PKC without any CCA2-conversion, as it was originally designed. Without such a conversion, the scheme will be vulnerable against adaptive chosen-ciphertext attacks [19]. However, a suitable conversion, like the Korbara-Imai-Conversion [11], will solve this problem. In Section 3.2 we show that the usage of a CCA2-conversion does not prevent the side channel attack described in Section 3.1.

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 [19] or [20]. In addition, $\mathbb{F}_{2^m}$ and the γ_i are public parameters.

Key Generation

The private key. The secret key consists of two parts. The first part of the secret key in the McEliece PKC is a Goppa polynomial $g(Y)$ of degree t over $\mathbb{F}_{2^m}$ according to definition 1, with random coefficients. The second part of the private key 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 $\mathbf{H}$ on the basis of $g(Y)$. 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).

Encryption. Assume Alice wants to encrypt a message $\boldsymbol{v} \in \mathbb{F}_2^k$. 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 $\boldsymbol{z} = \boldsymbol{v}\mathbf{G}^{\text{pub}} \oplus \boldsymbol{e}$.

Decryption. In order to decrypt the ciphertext, Bob computes $\boldsymbol{z}\mathbf{P}$. Then he applies error correction by executing an error correction algorithm, such as the Patterson Algorithm described in Section 2.3, to determine $\boldsymbol{e}\mathbf{P}$. Afterwards, he recovers the message $\boldsymbol{v}$ as the first k bits of $\boldsymbol{z} \oplus \boldsymbol{e}\mathbf{P}\mathbf{P}^{-1}$.

2.3 Error Correction for Irreducible Binary Goppa Codes

In the following we briefly describe how error correction can be performed with binary irreducible Goppa codes. The *error correction* of Goppa codes makes use of the so called error locator polynomial

$$\sigma_{\boldsymbol{e}}(X) = \prod_{j \in \mathcal{T}_e} (X - \gamma_j) \in \mathbb{F}_{2^m}[X], \tag{3}$$

where $\mathcal{T}_{\boldsymbol{e}} = \{i | e_i = 1\}$ and $\boldsymbol{e}$ is the error vector of the distorted code word to be decoded. Once the error locator polynomial is known, the error vector $\boldsymbol{e}$ is determined as

$$\boldsymbol{e} = (\sigma_{\boldsymbol{e}}(\gamma_0), \sigma_{\boldsymbol{e}}(\gamma_1), \cdots, \sigma_{\boldsymbol{e}}(\gamma_{n-1})) \oplus (1, 1, \cdots, 1). \tag{4}$$

The *Patterson Algorithm* is an efficient algorithm for the determination of the error locator polynomial. It can be found in detail in [19]. We will restrict our description to those features that are necessary to understand the attack we are going to present. Also, we do not provide derivations for most of the equations we specify in the following.

The Patterson Algorithm actually does not determine $\sigma_{\boldsymbol{e}}(X)$ as defined in Equation 3, but computes $\bar{\sigma}_{\boldsymbol{e}}(X) = \sigma_{\boldsymbol{e}}(X) \bmod g(X)$, where $\bar{\sigma}_{\boldsymbol{e}}(X) = \sigma_{\boldsymbol{e}}(X)$ if $\mathrm{wt}(\boldsymbol{e}) \leqslant t$.

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

$$\bar{\sigma}_e(X) = \alpha^2(X) + X\beta^2(X). \tag{5}$$

Defining $\tau(X) = \sqrt{S_{\boldsymbol{z}}^{-1}(X) + X} \bmod g(X)$, with $S_{\boldsymbol{z}}(X)$ being the syndrome of the distorted code word $\boldsymbol{z}$, the following equation holds:

$$\beta(X)\tau(X) = \alpha(X) \bmod g(X) \tag{6}$$

Then, assuming that no more than t errors occurred, Equation 6 can be solved by applying the Euclidean algorithm with a breaking condition concerning the degree of the remainder [19]. Specifically, the remainder in the last step is taken as $\alpha(X)$ and the breaking condition is $\deg(\alpha(X)) \leqslant \lfloor \frac{t}{2} \rfloor$. It can be shown that then, $\deg(\beta(X)) \leqslant \lfloor \frac{t-1}{2} \rfloor$.

From this, it follows that the polynomial $\bar{\sigma}_{\boldsymbol{e}}(X)$ defined over Equation 5 will be of degree $\leqslant t$. In the case that the number of errors is no larger than t, from Equation 3 it follows that $\deg(\bar{\sigma}_{\boldsymbol{e}}(X)) = \mathrm{wt}(\boldsymbol{e})$ since then $\bar{\sigma}_{\boldsymbol{e}}(X) = \sigma_{\boldsymbol{e}}(X)$

For the case of more than t errors, we give the following remark.

Remark 1. If $\mathrm{wt}(\boldsymbol{e}) > t$, then the $\deg(\bar{\sigma}_{\boldsymbol{e}}(X)) = t$ with probability $1 - 2^{-m}$.

This remark can be justified easily: Since the $\sigma_{\boldsymbol{e}}(X)$ computed via Equation 3 would yield $\deg(\sigma_{\boldsymbol{e}}(X)) = \mathrm{wt}(\boldsymbol{e})$, we find that the calculation $\bmod g(X)$ in Equation 6 leads to polynomials $\bar{\sigma}_{\boldsymbol{e}}(X)$ of degree t with coefficients that we can assume to be almost randomly distributed, where the leading coefficient is

not necessarily non zero. But clearly, for random coefficients out of $\mathbb{F}_{2^m}$, the probability that the leading coefficient is not zero is $1 - 2^{-m}$, which is amounts to 0.9995 for $m = 11$. Furthermore, experimental results confirm the claim of the remark.

3 Attack on the Degree of the Error Locator Polynomial

The dependence of the degree of the error locator polynomial $\bar{\sigma}_{\boldsymbol{e}}(X)$ on the number of errors in the decoding algorithm, which we examined in Section 2.3, can be used as a basis of a chosen-ciphertext side channel attack. We will describe it as a pure timing attack, though it clearly could be supported by incorporating analysis of the respective power traces.

3.1 The Timing Attack

When computing the error vector according to Equation 4, the error locator polynomial is evaluated 2^m times. Clearly, in a naive implementation, the time taken by the evaluation will increase with the degree of $\bar{\sigma}_{\boldsymbol{e}}(X)$.

The scenario for our attack is as follows: Alice encrypts a plaintext $\boldsymbol{v}$ to a ciphertext $\boldsymbol{z} = \boldsymbol{v}\mathbf{G}^{\text{pub}} \oplus \boldsymbol{e}$ according to the algorithm described in Section 2.2. Eve receives a copy of $\boldsymbol{z}$, and mounts the side channel attack by submitting manipulated ciphertexts $\boldsymbol{z}_i$ to Bob, who applies the decryption algorithm according to Section 2.2 to every single one of them. It is assumed that the decryption algorithm makes use of the Patterson Algorithm. Eve is able to measure the execution time of each decryption. In order to achieve a simple model, let us further assume that the only cause of timing differences is the evaluation of the error locator polynomial $\sigma_{\boldsymbol{e}}(X)$ in the Patterson Algorithm according to Equation 4.

The attack is described in algorithm 1. Here, sparse_vec (i) denotes the vector with zeros as entries except for the i-th position having value 1, and the first position being indexed by 0. The key idea is to flip the bit at position i in $\boldsymbol{z}$, resulting in $\boldsymbol{z}_i$, and then to find out whether the i-th position of $\boldsymbol{e}$ was zero or one. This in turn can be derived from the running time of the decryption algorithm on input $\boldsymbol{z}_i$, since $\bar{\sigma}_{\boldsymbol{e}}(X)$ will be of degree $t-1$ if $e_i = 1$, and of degree t otherwise.

3.2 The Timing Attack in the Presence of a CCA2-Conversion

A conversion like Pointcheval's [21] or Korbara and Imai's [11] makes sure that ciphertexts manipulated in the way described in algorithm 1 will not be decrypted, i.e. no plaintext will be output by the decryption device. This is ensured by a respective check performed after the error vector $\boldsymbol{e}$ has been determined via the Patterson Algorithm.

However, the possibility of our side channel attack is not affected by this fact, since in the presence of the conversion the attacker will still find a substring of the ciphertext which actually is equivalent to $\boldsymbol{z} = \boldsymbol{v}\mathbf{G}^{\text{pub}} \oplus \boldsymbol{e}$ and choose this as the target of his manipulations. Furthermore, the Patterson Algorithm will run

Algorithm 1. Timing Attack against the evaluation of $\sigma_{\boldsymbol{e}}(X)$

Require: ciphertext $\boldsymbol{z}$, and the parameter t, of the McEliece PKC.
Ensure: a guess $\boldsymbol{e}'$ of the error vector $\boldsymbol{e}$ used by Alice to encrypt $\boldsymbol{z}$.
1: **for** $i = 0$ to $n - 1$ **do**
2: Compute $\boldsymbol{z}_i = z \oplus \text{sparse_vec}\,(i)$.
3: Take the time u_i as the mean of N measured decryption times where $\boldsymbol{z}_i$ is used as the input to the decryption device.
4: **end for**
5: Put the t smallest timings u_i into the set M.
6: **return** the vector $\boldsymbol{e}'$ with entries $e'_i = 1$ when $u_i \in M$ and all other entries as zeros.

through all its steps regardless of whether the ciphertext has been manipulated or not. Only afterwards the algorithm will detect the manipulation and refuse decryption.

3.3 Implementation of the Attack

We realized the attack against a software implementation of McEliece. Specifically, our target was the implementation of the scheme in the FlexiProvider[1], which is a Java Cryptographic Extension (JCE) Provider. The implementation uses the Patterson Algorithm in the decoding step of the decryption phase. For simplicity, we did not include any CCA2-Conversion.

We executed the attack on an AMD Opteron 2218 CPU running at 2.6 GHz under Linux 2.6.20 and Java 6 from Sun. A single attack with $N = 2$ took less than 2 minutes, which makes it very effective and useable in a real world scenario. Even a remote attack against a TLS server using McEliece seems to be possible.

The security parameters we used for the attack are $m = 11$ and $t = 50$. The attack algorithm was realized just as depicted in algorithm 1. With the choice of $N = 2$ we recovered all positions of $\boldsymbol{e}$ correctly in half of the executed attacks. The exact results can be found in Appendix C.

3.4 Proposed Countermeasure

The reason for the comparatively high efficiency of the attack is that the error locator polynomial is evaluated 2^m times in the Patterson Algorithm. For the security parameter $m = 11$, as in our example, these are 2048 evaluations. This means that even a small difference in a single evaluation will be inflated to considerable size.

In order to avoid the differences in the decryption time arising from the different degrees of $\bar{\sigma}_{\boldsymbol{e}}(X)$, it is a straightforward countermeasure to simply raise its degree artificially in the case that it is found to be lower than t. Note that furthermore all coefficients in the polynomial of degree t have to be non zero in order to avoid timing differences.

[1] http://www.flexiprovider.de/

3.5 Improvements of the Attack

The simple version of the timing attack provided in algorithm 1 already enabled successful attacks under idealized conditions. For real life scenarios, two improvements of the attack are feasible.

- Once the attacker has found one position j with $e_j = 1$, he can apply an improved version of the attack. Specifically, he can then create the manipulated ciphertexts
$$\boldsymbol{z}'_i = \boldsymbol{z}_i \oplus \text{sparse_vec}\,(j)$$
for all $i \neq j$ and use them as input for the decryption device. As a result, each $\boldsymbol{z}'_i$ will contain either t or $t-2$ errors. Where in algorithm 1 the attacker had to distinguish between timings resulting from degrees of $\bar{\sigma}_{\boldsymbol{e}}(X)$ differing by 1, this difference in degrees is now 2, resulting in an even higher difference in the timings.
- In the attack given in Algorithm 1, it is already provided that the attacker takes the average time of multiple decryptions of the same ciphertext in order to decrease noise. Still, certain deterministic timing differences could arise in the algorithm, causing certain timings u_p and u_q to differ considerably, even though $e_p = e_q$.

  However, once the attacker knows a number of error and non-error positions, he can modify $\boldsymbol{z}_i$ in a way such that the number of errors remains constant. Each of these ciphertexts will contain the same number of errors with respect to the Goppa code as $\boldsymbol{z}_i$, but will cause the Patterson Algorithm to start with a different syndrome. Thus, if the attacker averages over the corresponding execution times, he can eliminate the possible timing differences arising from certain syndromes.

4 Other Side Channels

The McEliece system contains several other operations, which enable side channel attacks, if these operations are implemented in a straightforward manner, i.e. one-to-one according to the algorithm specification. In this section, two of these critical operations are presented: the setup of the parity check matrix $\mathbf{H}$ during key generation and the calculation of the matrix $\boldsymbol{z}\mathbf{P}$ during decryption. The first one presents a potential side channel for power attacks [14], the second one for cache attacks [22].

4.1 Generation of Parity Check Matrix

The parity check matrix $\mathbf{H}$ is generated by applying complex matrix operations over $\mathbb{F}_{2^m}$ based on the secret polynomial $g(Y)$. According to [19] an element of the check matrix $\mathbf{H}$ can be written as

$$h_{i,j} = g(\gamma_{j-1})^{-1} \sum_{s=t-i+1}^{t} g_s \gamma_{j-1}^{s-t+i-1}, \tag{7}$$

where $i = 1, \ldots, t$ and $j = 1, \ldots, n$ (see Appendix A).

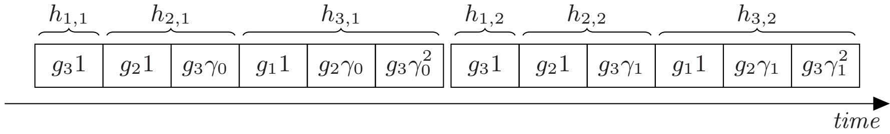

Fig. 1. Execution Order: Polynomial Multiplication

Inspecting this relation, two operations may be critical for power attacks [16]. These are the polynomial evaluation for the field elements $g(\gamma_j)$ and the multiplication of the polynomial coefficients with the powers of the field elements $g_s\gamma_{j-1}^{s-t+i-1}$.

Polynomial multiplication. Figure 1 shows schematically the multiplication steps executed to calculate the first and second column of $\mathbf{H}$. Here, we use $t = 3$ for simplicity. Remember that $\mathbf{H}$ has t rows and n columns.

From this figure it is evident that the multiplication steps and, thus, their power traces reveal high regularity. An exact application of the above relation results in multipliying g_3 by 1 once for each column of $\mathbf{H}$. Obviously, the power trace of these products may be used to indicate the start of the processing of a new column, which is essential for power attacks. Furthermore, it is highly probable that the power traces of $g_2\gamma_0$ and $g_2\gamma_1$ can be used to estimate the secret coefficient g_2 as the γ_i are public.

To complicate this attack, the multiplications $g_s\gamma_{j-1}$ must be performed in a manner, which does not leak information on g_s. This can be achieved (at least partially) by masking. Each g_s is multiplied by a random value $r_i \in \mathbb{F}_{2^m}$ before multiplying it by the field element γ_{j-1}. The de-masking using r_i^{-1} is performed after calculating the sum:

$$h_{i,j} = g(\gamma_{j-1})^{-1} r_i^{-1} \left(\sum_{s=t-i+1}^{t} (r_i g_s)\, \gamma_{j-1}^{s-t+i-1} \right). \tag{8}$$

In the above equation, the parentheses denote in which order the evaluation shall be performed.

This masking will be even more profitable if it is combined with a randomization of the order of term estimations. By this means the association of power traces with time is blurred considerably.

Polynomial evaluation. This operation is highly time-consuming and is performed in a pre-estimation phase, as a rule. The description in this section relates to this pre-estimation. Referring to the definition of the generator polynomial $g(Y)$, its evaluation for a field element γ_j can be written as $\sum_{i=0}^{t} g_i\gamma_j^i$. This means that polynomial evaluation amounts to multiplication over $\mathbb{F}_{2^m}$ with highly regular patterns, which again presents a possible side channel for power attacks. Fig. 2 depicts the chronological sequence of evaluating a polynomial of degree $t = 3$ for two field elements in a straightforward implementation. Similar to the case presented previously, countermeasures of masking and randomization should be employed.

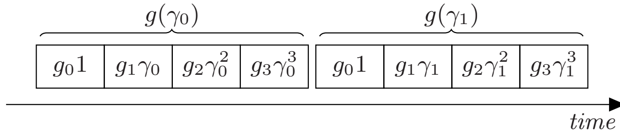

Fig. 2. Execution Order: Polynomial Evaluation

Using polynomial evaluation as power side channels is also possible in the decryption phase when the error vector is determined according to Equation 4.

4.2 Estimation of the Matrix $\boldsymbol{z}\mathbf{P}$

Presenting a possible power analysis attack scenario in Section 4.1, we will now focus upon a possible cache attack [22] scenario. Cache attacks, a specific type of so called microarchitectural attacks, have already been successfully mounted against software implementations [23].

The decryption of a ciphertext may also leak the other private key part, the permutation matrix $\mathbf{P}$. We assume that the permutation matrix itself is not stored directly as a matrix in the memory; it is rather implemented as some lookup-table for the rows and columns to save memory. This lookup-table is used in the decryption phase to compute $\boldsymbol{z}\mathbf{P}$ and $\boldsymbol{e}\mathbf{P}$.

In a straightforward implementation one may calculate these permutations by the following algorithm:

Algorithm 2. Permutation of $\boldsymbol{z}' = \boldsymbol{z}\mathbf{P}$

Require: Private permutation matrix $\mathbf{P}$ lookup-table $t^{\mathbf{P}}$ and ciphertext vector $\boldsymbol{z} \in \mathbb{F}_2^n$.
Ensure: The permutation $\boldsymbol{z}' = \boldsymbol{z}\mathbf{P}$.
1: **for** $i = 1$ to n **do**
2: Lookup $j = t_i^{\mathbf{P}}$.
3: Set $z_i' = z_j$.
4: **end for**
5: **return** permutated vector $\boldsymbol{z}'$.

The code in algorithm 2 will create memory access on addresses depending on the secret permutation $\mathbf{P}$. An attacker can use this to gain information about $\mathbf{P}$. Let us assume a scenario where the attacker has access to the system running the decryption process, and where the CPU of the computer supports simultaneous multithreading. The attacker executes a spy process parallel to the process of the decryption application. Let us further assume that the attacker knows the position of $\boldsymbol{z}$ in the main memory. Ideally, between any two iterations of the loop in algorithm 2, the spy process erases the content of $\boldsymbol{z}$ from the CPU cache and fills the respective cache blocks with some data of his own. It also regularly performs memory access to this data, measuring the execution time for this access.

From these timings, gathered while the decryption process was running in parallel, the attacker will be able to judge with certain precision which part of $\boldsymbol{z}$ was accessed during which iteration. Specifically, assume that for a certain iteration the time taken by the memory access of the spy process to a certain date indicates a cache miss. Then the attacker knows that the decryptions process accessed just that part of $\boldsymbol{z}$, which was stored in the same cache block. Note that the rule relating main memory addresses to cache blocks is system dependent and thus known to the attacker.

Due to the fact that in general the size of a cache block will be larger than one entry z_i, usually the attacker will not be able to get the exact index of the entry of $\boldsymbol{z}$ which has been accessed. Instead he will find out that for example an entry between z_0 and z_{31} must have been accessed. If the memory location of $\boldsymbol{z}$ differs in different executions and does not always have the same offset from the beginning of a cache block, the attacker might be able to narrow the access down to a single entry of $\boldsymbol{z}$.

In a weaker scenario, where the system running the decryption process does not support simultaneous multithreading, the attacker will not be able to peek into the decryption routine at every iteration, but with some probability the operating system will perform a context switch, interrupting algorithm 2 and continuing the spy process. In such a scenario the attack would be much harder, but still not impossible, assuming the attacker can repeat the measurement often enough.

Countermeasures. A possible contermeasure is to modify algorithm 2 to an algorithm whose memory access doesn't depend on the content of $t^{\mathbf{P}}$. We have implemented algorithm 3, which satisfies this requirement. It has constant running time, performs no jumps depending on secret input, and does only access memory addresses depending on public input. Therefore, it should be secure against timing-, cache-, and branch prediction attacks [24]. Unfortunatly, this increases the running time from $O(2^m)$ to $O((2^m)^2)$. Here, the operators $\sim$, &, >>, & =, |, and $-$ are used as they are used in the C programming language.

The idea behind algorithm 3 is the following: As in algorithm 2, $t_i^{\mathbf{P}}$ is read in line 2. Algorithm 2 would now read z_j and write it to z'_i. The write to z'_i is not critical, because i is public, but j depends on $\mathbf{P}$ and a read of z_j would reveal information about j and therefore about $\mathbf{P}$.

Algorithm 3 uses the following countermeasure. In line 3, z'_i is initialized with 0. In line 4, a new loop is started, where k runs from 0 to $n-1$. In every iteration, z'_i is read to l and z_k is read to m. Now, we have to distinguish between two cases:

1. $j = k$: In this case, we want to write $m = z_k = z_j$ to z'_i, as in algorithm 2.
2. $j \neq k$: In this case, we don't want to modify z'_i. But to create the same memory access as in case 1, we assign $l = z'_i$ to z'_i, and therefore leave z'_i unchanged.

In order to do this without an if-then-else statement, the following trick is used by algorithm 3: The XOR-difference s between j and k is computed in

Algorithm 3. secure permutation of $\boldsymbol{z}' = \boldsymbol{z}\mathbf{P}$

Require: Private permutation matrix $\mathbf{P}$ lookup-table $t^{\mathbf{P}}$ and ciphertext vector $\boldsymbol{z} \in \mathbb{F}_2^n$
Ensure: The permutation $\boldsymbol{z}' = \boldsymbol{z}\mathbf{P}$
1: **for** $i = 0$ to $n - 1$ **do**
2: $j = t_i^{\mathbf{P}}$
3: $z_i' = 0$
4: **for** $k = 0$ to $n - 1$ **do**
5: $l = z_i'$
6: $m = z_k$
7: $s = j \oplus k$
8: $s \mid= s >> 1$
9: $s \mid= s >> 2$
10: $s \mid= s >> 4$
11: $s \mid= s >> 8$
12: $s \mid= s >> 16$
13: $s \;\&= 1$
14: $s = \sim (s - 1)$
15: $z_i' = (s \& l) | ((\sim s) \& m)$
16: **end for**
17: **end for**
18: **return** $\boldsymbol{z}'$

line 7. If the difference is 0, $j = k$ and we are in case 1. If the difference is not 0, then we are in case 2.

Lines 8 to 14 now make sure, that if s is not 0, all bits in s will be set to 1. Now, the expression $(s \& l) | ((\sim s) \& m)$ will evaluate to l, and l will be written to z_i' in line 15.

If s was 0 after line 7, s will still be 0 after line 14. Now the expression $(s \& l) | ((\sim s) \& m)$ will evaluate to m, and m will be written to z_i' in line 15.

5 Conclusion

In this paper we have shown that the McEliece PKC like most known public key cryptosystems, bears a high risk of leaking secret information through side channels if the implementation does not feature appropriate countermeasures. We have detailed a timing attack, which was also implemented and executed against an existing software implementation of the cryptosystem. Our results show the high vulnerability of an implementation without countermeasures.

Furthermore, we presented a feasible power attack against the key generation phase, where certain operations involve the same secret value repeatedly. In general, key generation is a more difficult target for a side channel attack than decryption, because in contrast to that operation the attacker can only perform one measurement. But our considerations show, that without countermeasures, an implementation of the key generation might be vulnerable to a sophisticated power attack.

The cache attack designed to reveal the permutation that is part of the secret key, again benefits from the fact that the number of measurements the attacker may perform is in principle without any restraint. Thus the proposed secure algorithm seems to be an important countermeasure for software implementations intended for use in a multi user operating system.

Clearly, other parts of the cryptosystem require to be inspected with the same accuracy. This is especially true for the decryption phase, where the secret Goppa polynomial is employed in different operations.

The McEliece PKC, though existing for 30 years, has not experienced wide use so far. But since it is one of the candidates for post quantum public key cryptosystems, it might become practically relevant in the near future. With our work, besides the specific problems and solutions we present, we want to demonstrate that with the experience gathered in recent work exposing the vulnerabilities of other cryptosystems, it is possible to identify the potential side channels in a cryptosystem before it becomes commonly adopted.

References

1. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976)
2. Rivest, R.L., 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.S.: 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 Logarims. IEEE Transactions on Information Theory (1985)
5. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings, 35-th 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: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology, pp. 218–238 (1989)
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. Korbara, K., Imai, H.: Semantically secure McEliece public-key cryptosystems - conversions for McEliece PKC. In: Kim, K.-c. (ed.) PKC 2001. LNCS, vol. 1992. Springer, Heidelberg (2001)
12. Menezes, A., van Oorschot, P., Vanstone, S.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
13. Kocher, P.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology, pp. 104–113 (1996)

14. Kocher, P.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
15. Tsunoo, Y., Tsujihara, E., Minematsu, K., Miyauchi, H.: Cryptanalysis of Block Ciphers Implemented on Computers with Cache. In: International Symposium on Information Theory and Applications, pp. 803–806 (2002)
16. Schindler, W., Lemke, K., Paar, C.: A Stochastic Model for Differential Side Channel Cryptanalysis. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 30–46. Springer, Heidelberg (2005)
17. MacWilliams, F.J., Sloane, N.J.A.: The theory of error correcting codes. North-Holland, Amsterdam (1997)
18. Goppa, V.D.: A new class of linear correcting codes. Problems of Information Transmission 6, 207–212 (1970)
19. Engelbert, D., Overbeck, R., Schmidt, A.: A Summary of McEliece-Type Cryptosystems and their Security. Journal of Mathematical Cryptology (2006) (accepted for publication)
20. 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)
21. Pointcheval, D.: Chosen-chipertext security for any one-way cryptosystem. In: Imai, H., Zheng, Y. (eds.) PKC 2000. LNCS, vol. 1751, pp. 129–146. Springer, Heidelberg (2000)
22. Percival, C.: Cache missing for fun and profit, http://www.daemonology.net/papers/htt.pdf
23. Schindler, W., Acıiçmez, O.: A Vulnerability in RSA Implementations due to Instruction Cache Analysis and its Demonstration on OpenSSL. In: Malkin, T. (ed.) CT-RSA 2008. LNCS, vol. 4964, Springer, Heidelberg (2008)
24. Acıiçmez, O., Seifert, J.P., Koç, Ç.: Predicting secret keys via branch prediction. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377. Springer, Heidelberg (2007)

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

The parity check matrix $\mathbf{H}$ 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 $\operatorname{diag}(\ldots)$ denotes the diagonal matrix with entries specified in the argument. $\mathbf{H}$ 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:

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

The multiplication with $\left(Y^{t-1}, \cdots, Y, 1\right)^\top$ 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 [17].

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

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

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

C Experimental Results for the Timing Attack

Here, we show the experimentally determined probabilities (see Section 3.3) for the respective amounts of correctly guessed error positions.

	$N=1$	$N=2$
$\text{Prob}(\text{wt}(e' \oplus e) \leqslant 0)$	0%	48%
$\text{Prob}(\text{wt}(e' \oplus e) \leqslant 2)$	0%	77%
$\text{Prob}(\text{wt}(e' \oplus e) \leqslant 4)$	0%	96%
$\text{Prob}(\text{wt}(e' \oplus e) \leqslant 6)$	4%	99%
$\text{Prob}(\text{wt}(e' \oplus e) \leqslant 8)$	9%	99%
$\text{Prob}(\text{wt}(e' \oplus e) \leqslant 10)$	16%	100%
$\text{Prob}(\text{wt}(e' \oplus e) \leqslant 12)$	22%	100%
$\text{Prob}(\text{wt}(e' \oplus e) \leqslant 14)$	32%	100%
$\text{Prob}(\text{wt}(e' \oplus e) \leqslant 16)$	46%	100%
$\text{Prob}(\text{wt}(e' \oplus e) \leqslant 18)$	60%	100%
$\text{Prob}(\text{wt}(e' \oplus e) \leqslant 20)$	74%	100%
$\text{Prob}(\text{wt}(e' \oplus e) \leqslant 22)$	83%	100%
$\text{Prob}(\text{wt}(e' \oplus e) \leqslant 24)$	89%	100%