Side Channels in SW Implementation of the McEliece PKC

Marek Klein

***Abstract*—The McEliece cryptosystem is considered secure in the presence of quantum computers because there is no known quantum algorithm to solve the problem this cryptosystem is built on. However, naive implementation of the cryptosystem can open side channels, which can be used to gather information about the message or the secret key. In this paper we present results of chosen timing attacks on straightforward implementation of this cryptosystem. Furthermore, we present practical countermeasures and evaluate their efficacy.**

***Index Terms*—Side-channel attacks, timing attacks, post-quantum cryptography, code based cryptography, countermeasures.**

I. Introduction

PUBLIC key cryptography, or asymmetric cryptography, is a set of cryptographic algorithms that require two keys. One of the keys, public key, is published and everyone can use it in order to encrypt their secret. Although everybody knows how the message is encrypted, only the legitimate receiver, an owner of the second key, is able to decrypt the message. This property is widely used in the real world to secure financial transactions, to provide authenticity and in many other applications.

Security of currently most used cryptosystems, such as RSA [1], DSA or ECDSA [2], is based on the factorization of large primes or the calculation of the discrete logarithm. However, these cryptosystems are insecure in the case of existence of quantum computers, which are being actively developed these days. Therefore, several solutions have been proposed to be used instead of currently used cryptosystems. One of the candidates for post-quantum cryptography is the McEliece cryptosystem. It is based on the problem of decoding large linear codes without a visible structure. This problem belongs to the category of NP-complete problems and there is no known algorithm, solving this decoding problem in polynomial time.

In section II, we describe the McEliece cryptosystem, key generation, encryption and decryption.

In section III, we describe known attacks against the McEliece cryptosystem, and in section IV, we show that BitPunch implementation [3] is vulnerable to chosen timing side channel attacks and we present results of attacking chosen implementation.

In section V, we present practical countermeasures against chosen attacks and their efficiency.

Support by NATOs Public Diplomacy Division in the framework of ”Science for Peace”, Project MD.SFPP 984520, is acknowledged.
Manuscript received September 29, 2015; revised January 25, 2016.

II. The McEliece cryptosystem

The McEliece cryptosystem [5] was introduced by Robert J. McEliece in 1978. It is a public key cryptosystem based on linear codes. As one of the first cryptosystems, it used randomization during encryption. This cryptosystem uses error-correcting codes for which there exist fast decoding algorithms, for example Goppa codes.

In the following text, algorithms for generating keys, encryption and decryption are described.

A. Key Generation

Generation of the private and public key, according to [6], is described in 1. First, it is necessary to choose domain parameters m and t, where m defines the size of the finite field $\mathbb{F}(2^m)$ and t is the number of errors that can be corrected by the Patterson algorithm 4. Then monic irreducible polynomial $g(X) \in \mathbb{F}(2^m)$ of degree t is generated. Based on elements $\alpha_0, \dots, \alpha_{n-1}$, where $\alpha_i \in \mathbb{F}(2^m)$, and the polynomial $g(X)$, the control matrix $\mathbf{H}$ is created. The matrix $\mathbf{H}$ is computed as multiplication of matrices $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$, which are as follows:

$$\mathbf{X} = \begin{pmatrix} g_t & 0 & \cdots & 0 \\ g_{t-1} & g_t & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & \cdots & g_t \end{pmatrix}$$

$$\mathbf{Y} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_0 & \alpha_1 & \cdots & \alpha_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{t-1} & \alpha_1^{t-1} & \cdots & \alpha_{n-1}^{t-1} \end{pmatrix}$$

$$\mathbf{Z} = diag(g(\alpha_0)^{-1}, \dots, g(\alpha_{n-1})^{-1})$$

Afterward, a random permutation $\mathbf{P}$ is generated and the control matrix $\mathbf{H}$ is permuted by the inverse permutation $\mathbf{P}^T$. This permuted matrix is then transformed from the matrix over $\mathbb{F}(2^m)$ into the matrix $\mathbf{H}_2$ over $\mathbb{F}(2)$ where elements from $\mathbb{F}(2^m)$ are transformed into column vectors from $\mathbb{F}(2)$ of length m. From matrix $\mathbf{H}_2$, a generator matrix is created for a linear code and part of this matrix is published as a public key. Private key consists of permutation $\mathbf{P}$ and polynomial $g(X)$.

Algorithm 1 McEliece-PKC Key Generation.

```
Require: McEliece domain parameters $m$ and $t$.
Ensure: Public key $\mathbf{R}^T$ and private key $(\mathbf{P}, g(X))$.
1: Construct $\mathbb{F}(2^m) = \{\alpha_0, \ldots, \alpha_{n-1}\}$, where $n = 2^m$.
2: Generate a random monic, irreducible polynomial $g(X)$ of degree $t$, having coefficients in $\mathbb{F}(2^m)$ and $X \in \mathbb{F}(2^m)$.
3: Calculate the $t \times n$ control matrix $\mathbf{H}$ for the Goppa code generated by the polynomial $g(X)$.
4: Create a random $n \times n$ permutation matrix $\mathbf{P}$.
5: Calculate the permuted control matrix $\widehat{\mathbf{H}} = \mathbf{HP}^T$.
6: Transform the $t \times n$ matrix $\widehat{\mathbf{H}}$ over $\mathbb{F}(2^m)$ into the $mt \times n$ matrix $\mathbf{H}_2$ over $\mathbb{F}(2)$.
7: Bring $\mathbf{H}_2$ into the systematic form $\widehat{\mathbf{G}} = [\mathbb{I}_{mt}|\mathbf{R}]$.
8: The expanded public key is the $k \times n$ matrix over $\mathbb{F}(2)$, denoted as $\mathbf{G} = [\mathbf{R}^T|\mathbb{I}_k]$.
9: return $\mathbf{R}^T$ and $(\mathbf{P}, g(X))$
```

B. Encryption

Algorithm 2 describes the encryption process. The corresponding codeword $\mathbf{c}'$ from linear code, generated by the matrix $\mathbf{G}$, is computed for the message $\mathbf{m}$. This codeword is then encrypted by adding the error vector with t nonzero entries.

Algorithm 2 McEliece-PKC Encryption.

```
Require: $k$-bit plain text $\mathbf{m}$, public key $(\mathbf{G}, t)$.
Ensure: $n$-bit cipher text $\mathbf{c}$.
1: $\mathbf{c}' = \mathbf{mG}$
2: Generate the $n$-bit error vector $\mathbf{e}$ such that $hwt(\mathbf{e}) = t$.
3: $\mathbf{c} = \mathbf{c}' \oplus \mathbf{e}$
4: return $\mathbf{c}$
```

C. Decryption

Algorithm 3 describes decryption of the received message $\mathbf{c}$. The received message is permuted by the private permutation $\mathbf{P}$. Afterward, Patterson algorithm [4] is used to remove the error vector from the message and then the plain text is reconstructed.

Algorithm 3 McEliece-PKC Decryption.

```
Require: $n$-bit cipher text $\mathbf{c}$, private key $(\mathbf{P}, g(X))$.
Ensure: $k$-bit plain text $\mathbf{m}$.
1: Permute $\mathbf{c}$: $\mathbf{c}' = \mathbf{cP}$.
2: Use Patterson algorithm 4 to reconstruct the error vector $\mathbf{e}'$.
3: Permute the error vector $\mathbf{e} = \mathbf{e}'\mathbf{P}^T$.
4: Remove the error vector from the received message $\mathbf{c}' = \mathbf{c} \oplus \mathbf{e}$.
5: Reconstruct the plain text $\mathbf{m}$ from $\mathbf{c}'$.
6: return $\mathbf{m}$
```

III. TIMING SIDE-CHANNEL ATTACKS

There exist numerous different side-channel attacks on the McEliece cryptosystem. In this section, we describe such attacks, which we realized against the BitPunch implementation.

A. Attack against the Degree of the Error Locator Polynomial

Timing attack described in [7] can be executed during decryption of the received message. The attack is aimed at determining the error vector $\mathbf{e}$.

Let us assume we have a cipher text $\mathbf{c}$ and that we are looking for the corresponding plain text $\mathbf{m}$. The aim is to remove the error vector $\mathbf{e}$ from the received cipher text $\mathbf{c}$.

We try to decode the message $\mathbf{c}_i$ for all $\mathbf{e}_i$, where $i = 0, \ldots, n-1$.

During decryption it is necessary to determine the error $\mathbf{e}$ that was added to the message $\mathbf{c}'$. This error is determined by the error locator polynomial $\sigma_{\mathbf{c}}(X)$, whose degree is $deg(\sigma_{\mathbf{c}}) = hwt(\mathbf{e})$, if $hwt(\mathbf{e}) \leq t$. If $hwt(e) > t$, then $deg(\sigma_{\mathbf{c}}(X)) = t$ with probability $1 - 2^{-m}$. Therefore, evaluation time of the polynomial $\sigma(X)$ depends on the degree of this polynomial.

a) Attack description:

We only need to measure the time of evaluation of the error vector $\mathbf{e} = (\sigma_{\mathbf{c}_i}(\alpha_0), \ldots, \sigma_{\mathbf{c}_i}(\alpha_{n-1})) \oplus (1, \ldots, 1)$. As we can see, the polynomial $\sigma_{\mathbf{c}_i}(X)$ is evaluated n-times and if n is large enough then even a small difference in the degree of $\sigma_{\mathbf{c}_i}(X)$ might cause considerable time difference and therefore we can determine $\mathbf{e}$.

Let $\tau_i = T((\sigma_{\mathbf{c}_i}(\alpha_0), \ldots, \sigma_{\mathbf{c}_i}(\alpha_{n-1})) \oplus (1, \ldots, 1))$ be the time of decoding message $\mathbf{c}_i$. Put the t smallest τ_i into the set I. Then the error vector can be created as: $\mathbf{e} = \bigoplus_i \mathbf{e}_i$, for i such that $\tau_i \in I$.

b) Countermeasure:

To avoid this attack, we can artificially raise the degree of the polynomial $\sigma_{\mathbf{c}}(X)$ to t in case $deg(\sigma_{\mathbf{c}}(X)) < t$.

B. Timing Attack against Secret Permutation

Timing attack described in [8] can be used to determine the secret permutation $\mathbf{P}$. The attacker violates encryption schema by sending specific ciphertexts with only 4 errors instead of t errors.

To understand this attack, it is necessary to realize that the error locator polynomial $\sigma(X)$, determined during decryption, can be written in the following forms:

$$\sigma(X) = \sigma_t \prod_{j \in \varepsilon'} (X - \alpha_j) = \sum_{i=0}^{t} \sigma_i X^i \qquad (1)$$

where ε' is set of indexes i, for which $e'_i = 1$, i.e. those elements of $\mathbb{F}_{2^m}$ that correspond to error positions in the permuted error vector.

Authors use the ability of constructing their own cipher texts; therefore they can control the number of errors and positions of errors in the error vector $\mathbf{e}$. They decided to create an error vector $\mathbf{e}$ with the hamming weight $w < t$. Specifically, they used $w = 4$, since it is the only one offering a plain timing attack.

If $w = 4$, then $deg(\sigma(X)) = 4$. Since $deg(\sigma(X))$ is even, $deg(a(X))$ is 2. Hence, $a(X)$ provides a leading coefficient of $\sigma(X)$ and $deg(b(X)) \leq 1$. This freedom in the degree of $b(X)$ leads to two possible control flows in the decryption algorithm. One iteration in the Extended Euclidean algorithm (XGCD) (5) or zero iterations in the XGCD. These cases lead to two different forms of $\sigma(X)$. In case of one iteration, we find $\sigma_3 \neq 0$, because $b(X) = q_1(X)$. In case of zero iterations, we find $\sigma_3 = 0$, because $b(X) = 1$.

c) Attack description:

Let $\varepsilon = \{f_1, f_2, f_3, f_4\}$ be the set of indexes of four positions of errors in the non-permuted error vector $\mathbf{e}$.

We can rewrite the equation for the error locator polynomial (Equation 1) as

$$\sigma(X) = \sigma_4 \prod_{j\in\varepsilon} \left(X - \alpha_{\mathbf{P}_j}\right), \tag{2}$$

where $\mathbf{P}_j$ is the vector notation of permutation $\mathbf{P}$. While $\mathbf{e} = \mathbf{e}'\mathbf{P}$, we can write $e_i = e'_{\mathbf{P}_i}$ for entries of vector $\mathbf{e}$.

From Equation 2, we can write the coefficient σ_3 as a function of error positions:

$$\sigma_3(f_1, f_2, f_3, f_4) = \sigma_4\left(\alpha_{\mathbf{P}_{f_1}} + \alpha_{\mathbf{P}_{f_2}} + \alpha_{\mathbf{P}_{f_3}} + \alpha_{\mathbf{P}_{f_4}}\right). \tag{3}$$

The aim is to build a set of linear equations describing the secret permutation $\mathbf{P}$. Since the attacker can construct his own cipher texts with the hamming weight $hwt(\mathbf{e}) = 4$, he can ask the decryption device to decrypt his messages and measures the timing of step 4 of Patterson algorithm 4. If the attacker concludes from the timing that the number of iterations in XGCD algorithm is zero, the attacker adds equation $\sigma_3(f_1, f_2, f_3, f_4) = 0$ into the set of equations.

The equation system can be represented as an $l \times n$ matrix, where l is the number of equations and n is the length of the code used in the McEliece cryptosystem.

Depending on the rank of the matrix, a number of entries of the permutation have to be guessed.

d) Countermeasure:

As described in [8], to avoid this attack, it is necessary to check, and if needed, manipulate the degree of $\tau(X)$, because if the number of iterations in the XGCD algorithm 5 is zero, then $deg(\tau(X)) \leq d = \lfloor t/2 \rfloor$ before the first iteration.

It is necessary to perform the test whether $deg(\tau(X)) < d$ after determining $\tau(X)$ in Patterson algorithm 4. In case $deg(\tau(X)) < d$, then $\tau(X)$ must be manipulated in such a way that $deg(\tau(X)) = t - 1$.

It is recommended to use pseudo-random values derived from the cipher text to manipulate coefficients of $\tau(X)$. In case of using truly random values, the attacker might determine that the decryption operation is not deterministic.

e) Note:

Similar attack is described in [9]. The same as above, authors construct their own ciphertexts of low hamming weights and exploit the XGCD algorithm. Moreover, they also use error vectors of higher hamming weights; therefore, they can gather more linear equations.

IV. ATTACKS ON THE BITPUNCH LIBRARY

In this section we present results of attacking BitPunch implementation of the McEliece cryptosystem.

Attacks were realized on the following platform:

- CPU - Intel Core i5-3230M CPU @ 2.60GHz × 4
- Architecture - 64-bit
- Operating system - Ubuntu 14.04

In order to achieve the most accurate results, Intel Hyper-Threading, turbo mode, and frequency scaling were turned off. To measure execution time, we used the RDTSC instruction [10].

A. Attack against The Degree of ELP

In this section, we present results of attacks based on the degree of the error locator polynomial, described in subsection III-A.

First, we attacked the cryptosystem as in a real-life situation. We asked for decryption of manipulated ciphertext. For each ciphertext $\mathbf{c}_i = \mathbf{c} \oplus \mathbf{e}_i$, we measured ten times of the whole decryption process and averaged these iterations. With this simple attack, we were able to reveal from 45 to 50 errors. Results of the attack are presented in Figure 1.

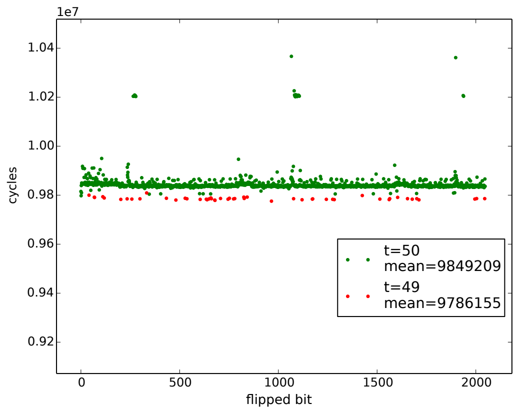

Fig. 1: Decryption times for ciphertext containing 50 error bits compared to decryption times for ciphertext containing 49 error bits.

In Figure 2, we can see decryption times corresponding to the ciphertext containing 50 error bits compared to decryption times corresponding to the ciphertext containing only 49 error bits. According to mean values and standard deviations, we can claim that the instance when the attacker revealed the bit in the error vector is easily distinguishable from when the attacker added the error bit to the ciphertext.

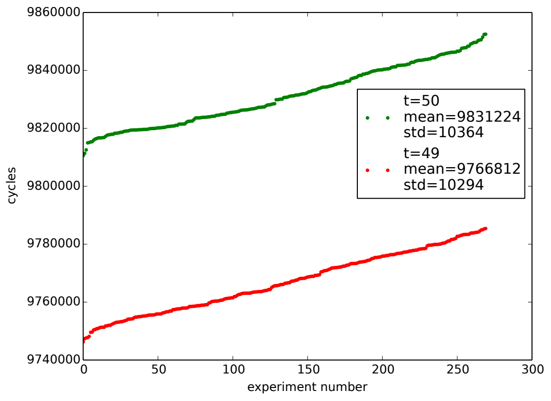

Fig. 2: Decryption times for ciphertext containing 50 error bits compared to decryption times for ciphertext containing 49 error bits.

B. *Attack against Secret Permutation and Syndrome Inversion*

In this section, we present results of attacks described in subsection III-B. However, to point out these time differences, we attacked the system with applied countermeasures described in section V.

This attack and the attack described in [9] are strongly tied, since they exploit the same vulnerability and time differences caused by revealing that $\sigma_3 = 0$ are added one to another. We can see this addition in Table I. This attack is required on the chosen platform for approximately 25 minutes to gain 102 equations like Equation 3, where $\sigma_3(f_1, f_2, f_3, f_4) = 0$.

	Permutation	Inversion	Decryption
$\sigma_3 \neq 0$	141169	160546	17637240
$\sigma_3 = 0$	60	95770	17416713
difference	141109	64776	220527

TABLE I: Attacks against permutation and inversion.

V. COUNTERMEASURES AGAINST ELP BASED ATTACK

In this section, we present countermeasures against attacks based on manipulation of error locator polynomial $\sigma(X)$ and their efficiency. In the following, we show the code causing timing differences in case of $t = 50$ compared to case of $t = 49$.

A. *Naive implementation*

In Code 1, we can see the naive implementation of determining error vector. Determination is done by evaluation of polynomial $\sigma(X)$ for each element from the support Γ. The critical operation is on line 3, which represents the evaluation of the element α_l, where $l = 0, \ldots, n-1$.

Code 1: Naive implementation.

```
1  ...
2  for (l = 0; l <
       ctx->code_spec->goppa->support_len;
       l++) {
3    tmp_eval = BPU_gf2xPolyEval(&sigma,
         ctx->math_ctx->exp_table[l],
         ctx->math_ctx);
4    if (tmp_eval == 0) {
5      BPU_gf2VecSetBit(error, l, 1);
6    }
7  }
8  ...
```

Since the aim of following countermeasures is to ensure constant execution time of Code 1, we provide graph of times needed to execute mentioned block of code in Figure 3.

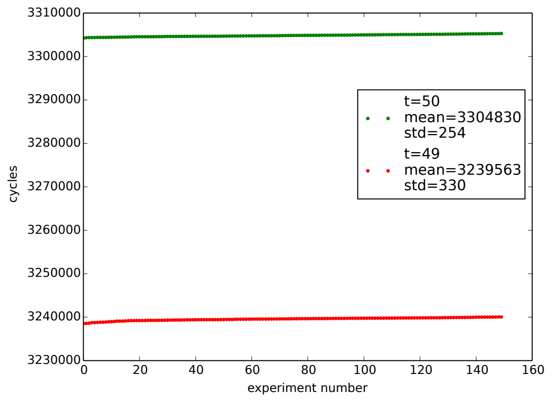

Fig. 3: Evaluation of $\sigma(X)$ without countermeasures.

B. *Practical countermeasures*

In Code 2, we can see the implementation of polynomial evaluation where running time directly depends on a degree of evaluated polynomial.

Code 2: Polynomial evaluation.

```
1  BPU_T_GF2_16x BPU_gf2xPolyEval(const
       BPU_T_GF2_16x_Poly *poly, const
       BPU_T_GF2_16x x, const BPU_T_Math_Ctx
       *math_ctx) {
2    int i;
3    BPU_T_GF2_16x ret = 0;
4    ret = poly->coef[0];
5
6    for (i = 1; i <= poly->deg; i++) {
7      ret = ret ^
           BPU_gf2xMulModT(poly->coef[i],
           BPU_gf2xPowerModT(x, i, math_ctx),
           math_ctx);
8    }
9    return ret;
10 }
```

Since the number of iterations in Code 2 depends on the degree of polynomial $\sigma(X)$, it is necessary to artificially increase its degree to the expected value. In this case, it is the number of errors that the decoding algorithm is capable of correcting. The degree of polynomial $\sigma(X)$ is increased at line 2 of Code 3.

However, just raising the degree of polynomial $\sigma(X)$ is insufficient. This significant time difference is caused by the *"If"* statement in line 4 of Code 1.

Code 3: Countermeasure 1.

```
1  ...
2  sigma.deg = ctx->t;
3  for (l = 0; l <
       ctx->code_spec->goppa->support_len;
       l++) {
4    tmp_eval = BPU_gf2xPolyEval(&sigma,
         ctx->math_ctx->exp_table[l],
         ctx->math_ctx);
5    BPU_gf2VecSetBit(error, l, !tmp_eval);
6  }
7  ...
```

Essential part of polynomial evaluation is multiplication of elements in the finite field. This is realized by using look-up tables, but as we can see in Code 4, this operation can differ according to its inputs. More specifically, if one of the elements a or b is 0, then the time needed to compute their product is shorter, because look-up tables are not used. This is caused by the *"if"* statement in line 3; when the condition is evaluated as true then 0 is returned immediately.

Code 4: Naive implementation of multiplication in finite field.

```
1  ...
2  BPU_T_GF2_16x BPU_gf2xMulModT(BPU_T_GF2_16x
       a, BPU_T_GF2_16x b, const
       BPU_T_Math_Ctx *math_ctx) {
3    int power;
4    if (a == 0 || b == 0)
5      return 0;
6    power = (math_ctx->log_table[a] +
         math_ctx->log_table[b]) %
         math_ctx->ord;
7    return math_ctx->exp_table[power];
8  }
9  ...
```

To avoid this difference, we decided to compute the product by look-up tables for every case and then find if one of the inputs is 0. This can be done by integer multiplication as is shown in Code Code 5, line 5. The result of the multiplication is saved in a new variable, which is, if needed, returned instead of the value computed by look-up tables.

Code 5: Multiplication in finite field with simple countermeasure.

```
1   ...
2   BPU_T_GF2_16x BPU_gf2xMulModT(BPU_T_GF2_16x
        a, BPU_T_GF2_16x b, const
        BPU_T_Math_Ctx *math_ctx) {
3     BPU_T_GF2_32x condition;
4     BPU_T_GF2_16x candidate;
5     int power;
6     power = (math_ctx->log_table[a] +
          math_ctx->log_table[b]) %
          math_ctx->ord
7     candidate = math_ctx->exp_table[power];
8     if (condition = (a * b))
9       return candidate;
10    return condition;
11  }
12  ...
```

After these modifications, time differences between evaluation times for polynomials of degree 50 and 49 are decreased, but it is still easy to distinguish between polynomials with 50 roots and polynomials with significantly less roots. It means that if attacker adds one more error to the ciphertext, then the polynomial $\sigma(X)$ is of degree 50, but it does not have 50 roots in a chosen finite field.

After the polynomial is evaluated, the appropriate bit is set to 1 if the result of evaluation is 0; otherwise, it is set to 0. This operation is executed by a macro shown in Code 6. We can see that setting the bit to 0 needs one more operation compared to setting the bit to 1.

Code 6: Naive implementation of bit setting macro.

```
#define BPU_gf2VecSetBit(v_pointer, i, bit)
if (bit) { \
  (v_pointer)->elements[(i) /
      (v_pointer)->element_bit_size] |=
      ((BPU_T_GF2) 1) << ((i) %
      (v_pointer)->element_bit_size);\
  } \
  else { \
    (v_pointer)->elements[(i) /
        (v_pointer)->element_bit_size] &=
        ((BPU_T_GF2) (0xFFFFFFFFu)) ^
        (((BPU_T_GF2) 1) << ((i) %
        (v_pointer)->element_bit_size));\
  }
```

Countermeasure shown in Code 7 not only provides the same number of operations, but also removes branching that can be used to attack the system by power analysis.

Code 7: Bit setting macro with countermeasure.

```
#define BPU_gf2VecSetBit(v_pointer, i, bit)
  (v_pointer)->elements[(i) /
      (v_pointer)->element_bit_size] &=
      ((BPU_T_GF2) (0xFFFFFFFFu)) ^
      (((BPU_T_GF2) 1) << ((i) %
      (v_pointer)->element_bit_size));\
  (v_pointer)->elements[(i) /
      (v_pointer)->element_bit_size] |=
      ((BPU_T_GF2) bit) << ((i) %
      (v_pointer)->element_bit_size);
```

Another operation used during the evaluation of polynomial $\sigma(X)$ is BPU_gf2xPowerModT(x, i, math_ctx). This operation is used to compute the ith power of the element $x \in \mathbb{F}(2^m)$. Execution time of this operation depends on parameters x and i. If one of these parameters is 0, then execution time is shorter. To avoid using this operation, we implemented polynomial evaluation by Horner's method, described by Equation 4, which uses only multiplication:

$$\sigma(X) = \sum_{i=0}^{n} a_i X^i. \tag{4}$$

After applying the previous countermeasures, the only more complex, thus the most vulnerable operation, is multiplication of elements $X, Y \in \mathbb{F}(2^m)$. When we look at its implementation in Code 5, we can see that the same number of instructions should be used during its execution. Nevertheless, different inputs a and b cause different execution times for this block of code, more specifically, modulo operation in line 5.

Logarithmic and exponential tables are implemented in the following way:
$E[i] = \alpha^i$, where $i = 0, \ldots, 2^m - 2$ and $E[2^m - 1] = 0$.
$L[E[i]] = i$, where $i = 0, \ldots, 2^m - 2$ and $L[0] = 2^m - 1$.
Since $D = 2^m - 1$ is used as a divisor, modulo operation needs more time if a dividend $L[a] + L[b] >= D$ than in case $L[a] + L[b] < D$. If $a = 0$ or $b = 0$, then $L[a] + L[b] >= D$; therefore, zero coefficients of $\sigma(X)$ cause this time difference.

In Code 8, modulo operation is replaced by the code at lines $6-10$. This replacement of modulo operation is not only a time constant, but also faster than the previous version. Unfortunately,it is not possible to apply this countermeasure on cryptosystem where parameter $n \neq 2^m$.

Code 8: With countermeasure 6.

```
...
BPU_T_GF2_16x BPU_gf2xMulModT(BPU_T_GF2_16x
     a, BPU_T_GF2_16x b, const
     BPU_T_Math_Ctx *math_ctx) {
  BPU_T_GF2_16x candidate;
  BPU_T_GF2_16x exp, bit, carry_mask = 1 <<
       math_ctx->mod_deg;
  BPU_T_GF2_32x condition;
  exp = math_ctx->log_table[a] +
       math_ctx->log_table[b];
  exp = exp + 1;
  bit = (exp & carry_mask);
  exp = (exp & math_ctx->ord);
  exp = (exp & math_ctx->ord) - !bit;
  candidate = math_ctx->exp_table[exp];
  if (condition = (a * b))
    return candidate;
  return condition;
}
...
```

In Figure 4, we can see that measured times for polynomials $\sigma(X)$ of degrees 50 and 49 are approximately the same.

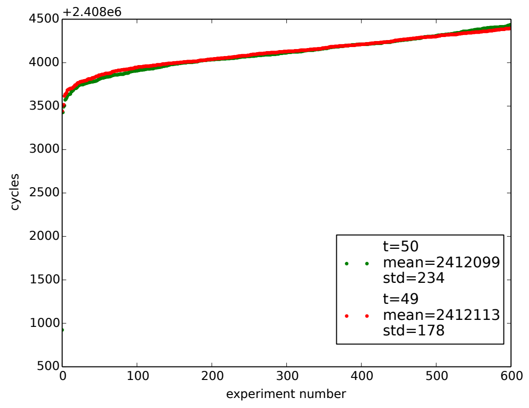

Fig. 4: Evaluation of $\sigma(X)$ - $deg(\sigma(X)) = 50$ compared to $deg(\sigma(X)) = 49$.

However, in Figure 5, it is shown that time differences between evaluation times for polynomials $\sigma(X)$ of degree 50 and 1 are still significant enough to say that algorithms are not time constant.

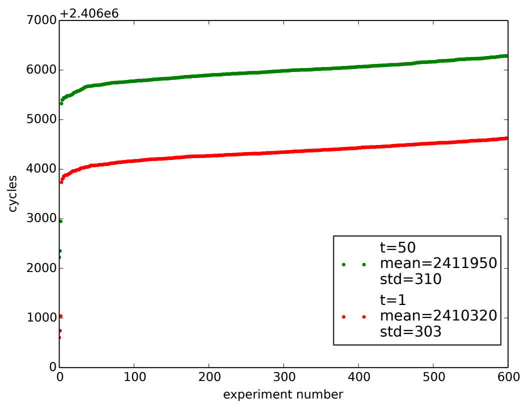

Fig. 5: Evaluation of $\sigma(X)$ - $deg(\sigma(X)) = 50$ compared to $deg(\sigma(X)) = 1$.

Since frequently used data can be stored in the CPU cache for faster access of processor to data, time differences pointed out in Figure 5 could be caused by this “caching”. We decided to replace multiplication done by logarithmic and exponential tables by time constant implementation of modular arithmetic as listed in Code 9. Unfortunately, this multiplication is approximately 2.5 times slower.

Code 9: Countermeasure 7.

```
BPU_T_GF2_16x BPU_gf2xMulModC(BPU_T_GF2_16x
     a, BPU_T_GF2_16x b, BPU_T_GF2_16x mod,
     BPU_T_GF2_16x mod_deg) {
  BPU_T_GF2_16x ret=0, i;
  for(i = 0; i < mod_deg; i++) {
    b ^= ((b >> mod_deg) & 1) * mod;
    ret ^= ((a >> i) & 1) * b;
    b = b << 1;
  }
  return ret;
}
```

In Figure 6, we can see that evaluation times for polynomials $\sigma(X)$ of degrees 50 and 49 are approximately the same. However, they are not exactly the same, but oscillate around the same values; see Table II.

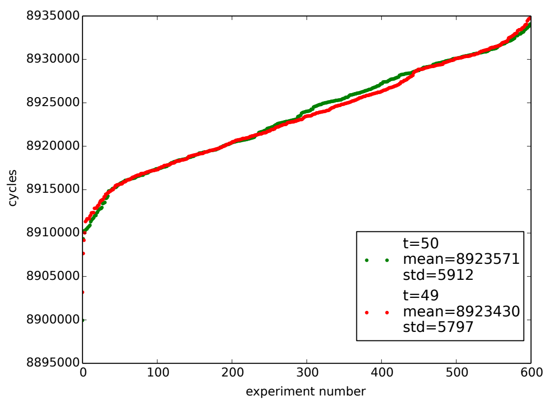

Fig. 6: Evaluation of $\sigma(X)$ - countermeasure no. 7.

	T_1	T_2	T_3	T_4
$deg(\sigma(X)) = 50$	8916516	8912341	8912524	8913037
$deg(\sigma(X)) = 1$	8917032	8911855	8912581	8913073
Difference	-516	486	-57	-36

TABLE III: Evaluation times of $\sigma(X)$ of degree 50 and 1.

VI. CONCLUSION

Proposed countermeasures should avoid the attack described in subsection III-A in a way in which it is not possible to distinguish if attacker guessed the correct position of bit in the error vector or not. On the other side, these countermeasures slow down the evaluation of polynomial $\sigma(X)$. This secured code needs 3 times longer time than naive implementation, where the biggest difference is caused by multiplication in finite field. This operation can be easily implemented in hardware; therefore, we suggest to construct a hybrid implementation of the McEliece cryptosystem. Hybrid implementation could use hardware implementation of time critical operations and software implementation of higher logic.

REFERENCES

[1] Rivest R. L., Shamir A., and Adleman L., "A method for obtaining digital signatures and public-key cryptosystems," Communications of the ACM, 1978, pp. 120-126.

[2] National Institute of Standards and Technology, "FIPS PUB 186-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS)," 2013.

[3] Gulyás A., Klein M., Kudláč J., Machovec F., and Uhrecký F., "Reálna implementácia code-based cryptography," Unpublished master's project, Slovak University of Technology in Bratislava, Slovakia, 2014.

[4] Patterson N., "The algebraic decoding of Goppa codes," IEEE Transactions on Information Theory 21, 2, 1975, pp. 203-207.

[5] McEliece R. J., "A public-key cryptosystem based on algebraic coding theory," DSN progress report, Vol. 42-44., 1978, pp. 114-116.

[6] Shoufan A., et al., "A novel processor architecture for McEliece cryptosystem and FPGA platforms," In Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP '09), IEEE Computer Society, 2009, pp. 98-105.

[7] Strenzke F., Tews E., Molter H. G., Overbeck R., and Shoufan A., "Side channels in the mceliece PKC," In Proceedings of the Second International Workshop, Post-Quantum Cryptography, 2008, pp. 216-229.

[8] Strenzke F., "A timing attack against the secret permutation in the mceliece PKC," In Proceedings of the Third international conference on Post-Quantum Cryptography (PQCrypto'10), Springer-Verlag, Berlin, Heidelberg, 2010, pp. 95-107.

[9] Strenzke F., "Timing attacks against the syndrome inversion in code-based cryptosystems," In Proceedings of the Fifth International Conference on Post-Quantum Cryptography - PQCrypto 2013, pp. 217-230.

[10] Paoloni G., "How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures," White Paper, 2010.

APPENDIX A ALGORITHMS

Algorithm 4 Patterson Algorithm.

```
Require: $n$-bit word **c**, Goppa polynomial $g(X)$.
Ensure: $n$-bit error vector **e**.
1: Compute syndrome polynomial $S_{\mathbf{c}}(X) = \mathbf{c}\mathbf{H}^T\left(X^{t-1},\ldots,X,1\right)^T$, where **H** is control matrix for Goppa code generated by polynomial $g(X)$.
2: Invert $S_{\mathbf{c}}^{-1}(X)$.
3: Let $\tau(X) = \sqrt{S_{\mathbf{c}}^{-1}(X) + X}$.
4: Find polynomials $a(X)$ and $b(X)$, so that $b(X)\tau(X) = a(X) \mod g(X)$, and $deg(a) \le \lfloor \frac{t}{2} \rfloor$.
5: Determine error locator polynomial $\sigma(X) = a^2(X) + xb^2(X)$, where $deg(\sigma) \le t$.
6: Reconstruct the error vector $\mathbf{e} = (\sigma(\alpha_0),\ldots,\sigma(\alpha_{n-1})) \oplus (1,\ldots,1)$.
7: return e
```

Algorithm 5 Extended Euclidean Algorithm.

```
Require: $\tau(X), g(X), d_{break}$
Ensure: $a(X), b(X)$ such that $b(X)\tau(X) = a(X) \mod g(X)$ and $deg(a) \le d_{break}$
1: $r_{-1}(X) = g(X)$
2: $r_0(X) = \tau(X)$
3: $b_{-1}(X) = 0$
4: $b_0(X) = 1$
5: $i = 0$
6: while $deg(r_i) > d_{break}$ do
7:     $i = i + 1$
8:     $q_i(X) = r_{i-2}(X) / r_{i-1}(X)$
9:     $r_i(X) = r_{i-2}(X) \mod r_{i-1}(X)$
10:    $b_i(X) = b_{i-2}(X) + q_i(X) b_{i-1}(X)$
11: $a(X) = r_i(X)$
12: $b(X) = b_i(X)$
13: return $a(X)$ and $b(X)$
```

Marek Klein received his Bc. degree in Modeling and Simulation of Event Systems and Ing. degree in Security of Information Technologies from Slovak University of Technology in Bratislava in 2013 and 2015 respectively. He currently works as developer at Disig, a.s. in the Department of Experimental Development.