SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Registration number: FEI-5384-5963

SIDE CHANNELS IN SW IMPLEMENTATION OF THE MCELIECE PKC

DIPLOMA THESIS

Study programme:	Applied Infromatics
Study field number:	2511
Study field:	9.2.9 Applied Informatics
Training workplace:	Institute of Computer Science and Mathematics
Thesis supervisor:	doc. Ing. Pavol Zajac, PhD.

2015 Bc. Marek Klein

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2014/2015
Evidenčné číslo: FEI-5384-5963

STU
FEI

ZADANIE DIPLOMOVEJ PRÁCE

Študent:	**Bc. Marek Klein**
ID študenta:	5963
Študijný program:	Aplikovaná informatika
Študijný odbor:	9.2.9. aplikovaná informatika
Vedúci práce:	doc. Ing. Pavol Zajac, PhD.
Miesto vypracovania:	Ústav informatiky a matematiky

Názov práce: **Postranné kanály v SW implementácii McElieceovho kryptosystému**

Špecifikácia zadania:

McElieceov kryptosystém je jedným zo základných predstaviteľov postkvantovej kryptografie.

Úlohy:
1. Naštudujte problematiku (softvérových) postranných kanálov s dôrazom na McElieceov kryptosystém.
2. Naštudujte a navrhnite protiopatrenia voči vybraným útokom.
3. Implementujte a otestujte zvolené útoky/protiopatrenia.

Zoznam odbornej literatúry:

1. Menezes, A J. – Oorschot, P C. – Vanstone, S A. *Handbook of applied cryptography.* Boca Raton: CRC Press, 1997. 780 s. ISBN 0-8493-8523-7.
2. McEliece, Robert J. "A public-key cryptosystem based on algebraic coding theory." DSN progress report 42.44 (1978): 114-116.
3. Shoufan, Abdulhadi, et al. "A novel processor architecture for McEliece cryptosystem and FPGA platforms." Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on. IEEE, 2009.

Riešenie zadania práce od:	22. 09. 2014
Dátum odovzdania práce:	22. 05. 2015

SLOVENSKÁ TECHNICKÁ UNIVERZITA
Fakulta elektrotechniky a informatiky
Ústav informatiky a matematiky
812 19 BRATISLAVA
L. S.

[illegible]

Bc. Marek Klein
študent

[illegible]

prof. RNDr. Otokar Grošek, PhD.
vedúci pracoviska

[illegible]

prof. RNDr. Otokar Grošek, PhD.
garant študijného programu

ABSTRAKT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Diplomová práca:	Postranné kanály v SW implementácii McElieceovho kryptosystému
Autor:	Bc. Marek Klein
Vedúci záverečnej práce:	doc. Ing. Pavol Zajac, PhD.
Miesto a rok predloženia práce:	Bratislava 2015

McEliece kryptosystém je považovaný za bezpečný voči útokom kvantovým počítačom, keďže nie je známy algoritmus, ktorý by bol schopný vyriešiť problém, na ktorom je postavená jeho bezpečnosť. Naivná implementácia tohto kryptosystému však môže byť zraniteľná voči útokom postrannými kanálmi. Tieto postranné kanály sa dajú využiť na získanie informácie o kľúči alebo odosielanej správe. V práci prezentujeme výsledky útokov na naivnú implementáciu McElieceovho kryptosystému, implementujeme praktické protiopatrenia a vyhodnocujeme ich úspešnosť.

Kľúčové slová: Útoky postrannými kanálmi, časové útoky, post-kvantová kryptografia, kryptografia založená na teórii kódovania, protiopatrenia.

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Diploma Thesis:	Side Channels in SW Implementation of the McEliece PKC
Author:	Bc. Marek Klein
Supervisor:	doc. Ing. Pavol Zajac, PhD.
Place and year of submission:	Bratislava 2015

The McEliece cryptosystem is considered secure in 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 secret key. In this thesis we present results of chosen timing attacks on straightforward implementation of this cryptosystem. Furthermore, we present practical countermeasures and evaluate their efficiency.

Key words: Side-channel attacks, timing attacks, post-quantum cryptography, code based cryptography, countermeasures.

Affidavit

I hereby declare that this master thesis has been written only by the undersigned and without any assistance from third parties. Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

Bratislava 17.5.2015

....

Marek Klein

Acknowledgment

I would like to express my gratitude to my supervisor Pavol Zajac for the useful comments, remarks and engagement through the learning process of this master thesis.

Contents

List of Algorithms

List of Figures

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 it. This property is widely used in real world to secure financial transactions, to provide authenticity and in many other applications.

Security of most currently used cryptosystems, such as RSA [7], 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 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 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.

Since this McEliece cryptosystem has significant drawbacks e.g., big public key, it was not used in real life. Therefore the need for studying its security is essential.

In this thesis we analyze vulnerabilities in BitPunch implementation [1] of the McEliece cryptosystem. This BitPunch library was implemented as a part of school project by A. Gulyás, M. Klein, J. Kudláč, F. Machovec and F. Uhrecký.

In section 1 we describe the McEliece cryptosystem, key generation, encryption and decryption. Furthermore, we summarize existing side-channel attacks against the McEliece cryptosystem and proposed countermeasures.

In section 2 we show that BitPunch implementation [1] is vulnerable to these attacks and we present results of attacking chosen implementation.

In section 3 and section 4 we present practical countermeasures against chosen attacks and their efficiency.

This thesis was realised as a part of the project "Secure implementation of post-quantum cryptography", NATO Science for Peace and Security
Programme Project Number: 984520.

1 Analysis

1.1 Notation

Notations used in this thesis are described in the following:
$\mathbf{c}$ - received message of length n
$\mathbf{e}$ - error vector of length n and $hwt(\mathbf{e}) = t$
$\mathbf{e}_i$ - vector of length n, with only one nonzero entry at i-th position
$\mathbf{m}$ - plain text corresponding to the message $\mathbf{c}$
$\mathbf{c}_i$ - vector such that: $\mathbf{c}_i = \mathbf{c} \oplus \mathbf{e}_i$
$\mathbf{G}$ - generator matrix
$\mathbf{H}$ - parity-check matrix
α - element in a finite field
$\mathbb{I}_k$ - $k \times k$ identity matrix
$hwt(\mathbf{v})$ - hamming weight of vector $\mathbf{v}$
$\mathbb{F}_{2^m}$ - finite field of order 2^m.

1.2 Preliminaries

The McEliece cryptosystem, which is discussed in this thesis, is based on error-correcting linear codes. This section provides brief introduction to coding theory, error-correcting codes and defines basic terms used in this thesis.

One of the basic aims of coding theory is to find solution how to transmit data over an unreliable channel, which results in errors in the sent message. For this purpose error-correcting codes are used. For these codes there exist efficient decoding algorithms capable of removing certain number of errors in transmitted message.

Definition 1.1 *Hamming weight of the element $\boldsymbol{c} \in \mathbb{F}_2^n$ is the number of nonzero entries of $\boldsymbol{c}$.*

Definition 1.2 *Binary (n, k, d) code $\mathcal{C}$ is a binary linear code of length n and dimension k, therefore k-dimensional subspace of $\mathbb{F}_2^n$, where every element's length is n and if $\boldsymbol{a}, \boldsymbol{b} \in \mathcal{C}$ and $c \in \mathbb{F}_2$, then $\boldsymbol{a} + \boldsymbol{b} \in \mathcal{C}$ and $c.\boldsymbol{a} \in \mathcal{C}$. Additionally, $hwt(\boldsymbol{a} + \boldsymbol{b}) \geq d$.*

The McEliece cryptosystem, as it was originally proposed in [4], is based on irreducible Goppa codes. Patterson's algorithm [6], described in alg. 9, can be used for fast decoding of irreducible Goppa codes.

Definition 1.3 *Let the polynomial $g(X) = \sum_{i=0}^{t} g_i X^i \in \mathbb{F}_{2^m}[X]$ be monic and irreducible over $\mathbb{F}_{2^m}[X]$. Let m and t be positive integers. Then $g(X)$ is called a Goppa polynomial. Then an irreducible binary Goppa code is defined as $\mathcal{C}(g(X), \Gamma) = \{\boldsymbol{c} \in \mathbb{F}_2^n : S_c(X) = \sum_{i=0}^{n-1} \frac{c_i}{X - \alpha_i} = 0 \mod g(X)\}$, where $n \leq 2^m$, $S_c(X)$ is the syndrome of $\boldsymbol{c}$, $\Gamma = (\alpha_i : i = 0, \ldots, n-1)$ is the support of the code, where α_i are pairwise distinct elements of $\mathbb{F}_{2^m}$, and c_i are entries of the vector $\boldsymbol{c}$.*

1.3 The McEliece cryptosystem

The McEliece cryptosystem [4] 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 exist fast decoding algorithm e.g., Goppa codes.

At present, standardly used cryptosystems based on elliptic curves and factorization problems might be broken in case of existence of quantum computer. Since there is no known algorithm providing fast decoding for randomly looking linear codes, this cryptosystem is a good candidate for post-quantum cryptography.

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

1.3.1 Key Generation

Generation of the private and public key is described in alg. 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 which is possible to correct by Patterson alg. 9. Then monic irreducible polynomial $g(X) \in \mathbb{F}(2^m)[X]$ of degree t is generated. Based on elements $\alpha_0, \ldots, \alpha_{n-1}$, 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 look as follows:

$$\mathbf{X} = \begin{pmatrix} g_t & 0 & \cdots & 0 \\ g_{t-1} & g_t & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & \cdots & g_t \end{pmatrix} \mathbf{Y} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_0 & \alpha_1 & \cdots & \alpha_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{t-1} & \alpha_1^{t-1} & \cdots & \alpha_{n-1}^{t-1} \end{pmatrix} \mathbf{Z} = diag(g(\alpha_0)^{-1}, \ldots, g(\alpha_{n-1})^{-1})$$

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 $\widehat{\mathbf{H}} = \mathbf{H}\mathbf{P}^T$.
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 $\mathbf{G} = \left[\mathbf{R}^T|\mathbb{I}_k\right]$.
9: return $\mathbf{R}^T$ and $(\mathbf{P}, g(X))$
```

1.3.2 Encryption

Algorithm 2 describes 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 $\mathbf{c}$
```

1.3.3 Decryption

Algorithm 3 describes decryption of the received message **c**. The received message is permuted by the private permutation **P**. Afterward, Patterson alg. 9 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 alg. 9 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 $\mathbf{m}$
```

1.4 Side Channels

In traditional cryptographic models the communication between Alice and Bob is secured by some mathematical function. In theoretical model adversary knows this function, and some plain and ciphertext pairs. Security proofs involve only these components but in real world there are many other aspects which can make this system vulnerable to some different than pure mathematical attacks, e.g. side-channel attacks.

Side channels are physically observable processes happening in a software or hardware implementation of a cryptosystem. Based on the type of information leakage, side-channel attacks can be divided into the following main categories:

- **Timing Attacks**, where different input data might cause different running time of an algorithm, and thus reveal some information about the secret key or the message itself.
- **Power Consumption Attacks**. Currently most of cryptographic devices are implemented in CMOS logic. Such circuits changes states of its gates with every clock. This leads to charging and discharging capacitors what influences current flow, which is easily measurable at the outside of the device.
- **Electromagnetic Radiation Attacks** are similar to previous power consumption attacks, since charging and discharging of capacitors not only influences current flow, but also generates certain electromagnetic field.

- **Error Message Attacks** can target devices which implement a decryption scheme. In case there is feedback from the device whether the message was successfully decrypted or not and the attacker knows the reason why decryption failed, he can reveal some information by sending well chosen ciphertexts into the device and observing the reaction.

1.5 Existing attacks

There exist numerous different side-channel attacks on the McEliece cryptosystem. In this section we describe attacks, which we want to prevent by implementing efficient countermeasures.

1.5.1 Reaction attack

Reaction attack described in [3], can be executed during decryption of the received message. The attack is aimed at determining the error vector $\mathbf{e}$.

This attack belongs to the category of chosen-ciphertext attacks, thus authors assume that the attacker can freely ask to decrypt an arbitrary message. To make this attack harder, authors decided to use decryption device which gives only information about successfully decrypted message or about failing decryption.

The attack rests upon one premise:
If an (n, k, d) error-correcting code is used which can correct $t = \lfloor \frac{d-1}{2} \rfloor$ errors, then a decoder will not attempt to correct a vector which has more than t errors.

Attack description

Let us assume we have a cipher text $\mathbf{c}$ and 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 $i = 0, \ldots, n-1$. If the decoder fails, the message $\mathbf{c}_i$ contains more than t errors, thus error vector $\mathbf{e}$ does not contain 1 at the i-th position. If the message is successfully decoded, then the error vector $\mathbf{e}$ contains 1 at the i-th position and we can add value i into the set I. Then error vector can be created as: $\mathbf{e} = \bigoplus_i \mathbf{e}_i, i \in I$.

Algorithm 4 Reaction attack

```
Require: Cipher text $\mathbf{c}$, McEliece parameters ($t$, $n$)
Ensure: Error vector $\mathbf{e}$
1: $\mathbf{e} = (0, \ldots, 0)$
2: for $i = 0$ to $n-1$ do $\mathbf{c}_i = \mathbf{c} \oplus \mathbf{e}_i$
3:     Ask for decryption of $\mathbf{c}_i$.
4:     if Message decoded then
5:         Put $i$ into the set $I$.
6: for $i \in I$ do
7:     $\mathbf{e} = \mathbf{e} \oplus \mathbf{e}_i$
8: return $\mathbf{e}$
```

Countermeasure

This attack is easy to avoid by using CCA2-conversion scheme proposed by authors in [3].

This scheme provides message integrity and in case of data corruption the decoder does not try to decode message even if the vector $\mathbf{e}_i$ contains 1 at the same position as the vector $\mathbf{e}$.

1.5.2 Attack against the Degree of the Error Locator Polynomial

Timing attack described in [11], can be executed during decryption of the received message. The attack is aimed at determining the error vector $\mathbf{e}$.

As in section 1.5.1, authors describe the chosen-cipher text attack, but now the CCA2-conversion can not avoid the attack.

During decryption it is necessary to determine, what error $\mathbf{e}$ 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.

Attack description

The structure of the attack is the same as in section 1.5.1, but now we do not wait for the answers from the decryption device. 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 small difference in the degree of $\sigma_{\mathbf{c}_i}(X)$ might cause considerable time difference and thus 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 $\tau_i \in I$.

Algorithm 5 Timing attack against degree of the error locator polynomial

```
Require: Ciphertext $\mathbf{c}$, McEliece parameters $(t, n)$
Ensure: Error vecotr $\mathbf{e}$
 1: $\mathbf{e} = (0, \ldots, 0)$
 2: for $i = 0$ to $n - 1$ do
 3:     $\mathbf{c}_i = \mathbf{c} \oplus \mathbf{e}_i$
 4:     Measure the decryption time $\tau_i$ for the cipher text $\mathbf{c}_i$.
 5:     Put $\tau_i$ into the set $L$.
 6: Put the $t$ smallest $\tau_i$ from the set $L$ into the set $I$.
 7: for $k = 0$ to $t - i$ do
 8:     Get the index $i$ of the $k$-th element of $I$.
 9:     $\mathbf{e} = \mathbf{e} \oplus \mathbf{e}_i$
10: return $\mathbf{e}$
```

Countermeasure

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

1.5.3 Timing Attack against Patterson Algorithm

To describe this attack it is necessary to be familiar with Patterson alg. 9 and extended Euclidean alg. 8.

The timing attack described in [8] can be executed during establishing of the error locator polynomial $\sigma(X)$ and is aimed at determining the error vector $\mathbf{e}$.

The error locator polynomial is established by polynomials $a(X)$ and $b(X)$ which are found by extended Euclidean alg. 8, in step 4 of Patterson alg. 9. The attack is based on the fact that the number of iterations in extended Euclidean algorithm depends on the Hamming weight of the error vector $\mathbf{e}$.

As we can see, it is possible to exploit this side channel irrespective to the countermeasure proposed in section 1.5.2 because raising the degree of polynomial $\sigma_{\mathbf{c}}(X)$ closes only the side channel in step 6 of Patterson algorithm but the side channel in step 4 remains opened.

Attack description

The attack varies depending on the number of errors t. For the simplicity we describe only the case when t is odd.

The structure of the attack is the same as in previous attacks described in section 1.5.1 and section 1.5.2. The attacker needs to measure the time of the XGCD. In case the attacker guesses the correct position of 1 in the error vector $\mathbf{e}$, then the iteration number of the XGCD is equal or less than $\frac{t-1}{2} - 1$ and the decryption time is slightly shorter.

Let $\tau_i = T(XGCD(\mathbf{c}_i))$ be the time of establishing polynomials $a(X)$ and $b(X)$ for the corresponding $\mathbf{c}_i$. 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 $\tau_i \in I$.

Algorithm 6 Timing attack against XGCD for odd t

```
Require: Cipher text $\mathbf{c}$, McEliece parameters $(t, n)$
Ensure: Error vector $\mathbf{e}$
 1: $\mathbf{e} = (0, \dots, 0)$
 2: for $i = 0$ to $n - 1$ do
 3:     $\mathbf{c}_i = \mathbf{c} \oplus \mathbf{e}_i$
 4:     Measure the decryption time $\tau_i$ for the cipher text $\mathbf{c}_i$.
 5:     Put $\tau_i$ into the set $L$.
 6: Put the $t$ smallest $\tau_i$ from the set $L$ into the set $I$.
 7: for $k = 0$ to $t - i$ do
 8:     Get the index $i$ of the $k$-th element of $I$.
 9:     $\mathbf{e} = \mathbf{e} \oplus \mathbf{e}_i$
10: return $\mathbf{e}$
```

In case of even t, this algorithm must be modified to support guessing at least two positions of ones in the error vector $\mathbf{e}$.

Countermeasure

To avoid this attack it is necessary to force the XGCD algorithm to run as many iterations as it would run in case of processing non-manipulated message. In other words, we need to enforce the continuation of XGCD until proper degrees of $a(X)$ and $b(X)$ are reached.

Specifically, if t is even, the degree of polynomial $a(X)$ can be used to determine the error vector. In case t is odd, then the degree of polynomial $b(X)$ can be used to determine the error vector. Continuation of the XGCD execution can be achieved by manipulating the degree of reminder polynomials $r_i(X)$, see steps 13 and 18 in alg. 7.

As we can see, this countermeasure also closes the side channel described in section 1.5.2.

Algorithm 7 Extended Euclidean Algorithm with countermeasure

```
Require: $\tau(X), g(X), d_{break}$
Ensure: $a(X), b(X)$ such that $b(X)\tau(X) = a(X) \mod g(X)$ and $deg(a) \leq d_{break}$
 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:     if $t$ even then
12:         if $deg(r_i(X)) < d_{break}$ then
13:             Manipulate $r_i(X)$, so that $deg(r_i) = deg(r_{i-1}) - 1$
14:     else
15:         if $deg(r_i(X)) \leq d_{break}$ AND $deg(b_i) < d_{break}$ then
16:             Manipulate $r_i(X)$, so that $deg(r_i) = deg(r_{i-1}) - 1$
17: $a(X) = r_i(X)$
18: $b(X) = b_i(X)$
19: return $a(X)$ and $b(X)$
```

1.5.4 Timing Attack against Secret Permutation

Timing attack described in [9] can be used to determine the secret permutation **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 realise that the error locator polynomial $\sigma(X)$, determined during decryption, can be written in the following forms:

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

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 different approach than attackers in section 1.5.1, section 1.5.2, and section 1.5.3. In these sections attacks where attackers manipulate cipher texts are described. In [9] authors use the ability of constructing their own cipher texts, thus they can control the number of errors and positions of errors in the error vector **e**. They decided to create an error vector **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. Thus $a(X)$ 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 XGCD (alg. 8) 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 $b(X) = 1$.

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 alg. 9. If the attacker concludes from the timing that the number of iterations in XGCD algorithm is zero, he 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 has to be guessed.

Countermeasure

To avoid this attack it is necessary to check and if needed then manipulate the degree of $\tau(X)$. Because if the number of iterations in the XGCD alg. 8 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 alg. 9. 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, attacker might determine that the decryption operation is not deterministic.

1.5.5 Timing Attack against Syndrome Inversion

Attack described in [10] can be used to gain information about the secret support. The same as the attack described in section 1.5.4, this attack is realised measuring the time of decryption for own cipher texts.

To understand this attack it is necessary to be familiar with the syndrome polynomial, which is defined as

$$S(X) = \sum_{i=1}^{w} \frac{1}{X \oplus \epsilon_i} = \frac{\Omega(X)}{\sigma(X)} \mod g(X) \tag{4}$$

where w is the hamming weight of the error vector $\mathbf{e}$ and the set $\{\epsilon_i | i \in \{1, \ldots, w\}\}$ is the set of support elements associated with indexes of bits in the error vector having value one. In order to decrypt the message it is needed to compute the inverse polynomial $S^{-1}(X)$ of the syndrome polynomial $S(X)$. This is done by Extended Euclidean alg. 8.

In [10] it is shown that the number of iterations M needed to compute $S^{-1}(X)$ in alg. 8 depends on degrees of polynomials $\Omega(X)$ and $\sigma(X)$ in the following way:

$$M \leq M_{max} = deg(\Omega(X)) + deg(\sigma(X)). \tag{5}$$

This relation between the degree of polynomial $\Omega(X)$ and the number of iterations in the XGCD allows the attacker to reveal information about particular coefficients of polynomial $\sigma(X)$. As in the attack described in section 1.5.4, authors create their own cipher text and use error vectors of weights $w = 4$, $w = 6$, $w = 1$.

In the case $w = 4$ the syndrome polynomial looks as follows:

$$S(X) = \frac{\Omega(X)}{\sigma(X)} = \sum_{i=1}^{4} = \frac{1}{X \oplus \epsilon_i} = \frac{\sigma_3 X^2 \oplus \sigma_1}{X^4 \oplus \sigma_3 X^3 \oplus \sigma_2 X^2 \oplus \sigma_1 X \oplus \sigma_0} \mod g(X) \tag{6}$$

where

$$\sigma_3 = \epsilon_1 \oplus \epsilon_2 \oplus \epsilon_3 \oplus \epsilon_4,$$

As we can see, the coefficient at the highest power of X of polynomial $\Omega(X)$ is given by σ_3. Based on Equation 5, we can see that in the case $\sigma_3 = 0$ the number of iterations in the XGCD alg. 8 is 4, otherwise it is 6. By measuring the time needed to compute the inverse syndrome polynomial $S^{-1}(X)$ it is possible to find $\sigma_3 = 0$ or $\sigma_3 \neq 0$.

In the case $w = 6$ the syndrome polynomial looks as follows:

$$S(X) = \frac{\Omega(X)}{\sigma(X)} = \frac{\sigma_5 X^4 \oplus \sigma_3 X^2 \oplus \sigma_1}{X^6 \oplus \sigma_5 X^5 \oplus \sigma_4 X^4 \oplus \sigma_3 X^3 \oplus \sigma_2 X^2 \oplus \sigma_1 X \oplus \sigma_0} \mod g(X) \tag{7}$$

where the attacker is interested in coefficients σ_3 and σ_5

$$\sigma_3 = \sum_{j=3}^{6} \sum_{k=1}^{j-1} \sum_{l=1}^{k-1} \epsilon_j \epsilon_k \epsilon_l, \tag{8}$$

$$\sigma_5 = \sum_{i=1}^{6} \epsilon_i. \tag{9}$$

If coefficients σ_3 and σ_5 are both 0 then $deg(\Omega(X)) = 0$. It means that the maximal number of iterations in the XGCD during computing the inverse syndrome polynomial is 6, otherwise in the case $\sigma_5 \neq 0$ the number of iterations is 10. Thus, the attacker is able to learn equations $\sigma_5 = 0$ and $\sigma_3 = 0$ by measuring the time of computing the inverse syndrome polynomial.

The vulnerability for $w = 1$ is in the polynomial division alg. 10 performed in step 8 of the XGCD alg. 8. In this case the inverse polynomial is always computed in one iteration but during the polynomial division it is possible to distinguish between the two cases. If $\epsilon_1 = 0$ then only one iteration during the division is needed, otherwise two iterations are needed. Thus the attacker can reveal the index z of the support element $\alpha_z = 0$.

Attack description

First step of the attack is to gain linear equations through $w = 4$ vulnerability. It is possible to build the system of linear equations with the rank $n - m - 1$. One more equation can be gained through the $w = 1$ vulnerability and added to the system. This system is brought into the row echelon form:

α_0	α_1	$\dots$	α_{n-m-3}	α_{n-m-2}	β_0	$\dots$	β_{m-1}
1	0	0	0	0	Y	$\dots$	Y
0	1	0	0	0	Y	$\dots$	Y
$\vdots$							
0	0	0	0	1	Y	$\dots$	Y

where $Y \in \{0, 1\}$.

The elements associated with the m rightmost columns must be a basis $\{\beta_i\}$. In this basis we know the representation of each α_i. In case the attacker knows the values of these basis elements he can easily compute all α_i.

Next step is to gain cubic equations through the $w = 6$ vulnerability. In order to achieve efficient guessing of values of β_i, positions of ones in error vectors and their corresponding values $\epsilon_i, i = 1, \dots, 6$ must fulfill specific conditions described below:

1. $\epsilon_i \in span(\{\beta_{s_1}, \beta_{g_1}, \beta_{g_2}, \beta_{g_3}\})$, where β-s are arbitrarily chosen basis elements and β_{s_1} is the one to be solved.
2. $\sum_{i=1}^{6} \epsilon_i = 0$.
3. Two of ϵ_i contain β_{s_1}. This condition leads to a quadratic equation for β_{s_1}.

Cipher texts constructed with these error vectors are then put into the decryption device and by measuring the time the attacker tries to find whether $deg(\Omega(X)) = 0$. After c_1 equations for β_{s_1} are found, where c_i is a parameter of the attack, the second set of equations is constructed for β_{s_2}. Now the difference is that $\beta_{s_2} \notin \{\beta_{s_1}, \beta_{g_1}, \beta_{g_2}, \beta_{g_3}\}$ and $\epsilon_i \in span(\{\beta_{s_1}, \beta_{g_1}, \beta_{g_2}, \beta_{g_3}, \beta_{s_2}\})$. In this manner $m - 3$ sets for different β_{s_i} are collected.

The last step of the attack is to guess initial values of elements $\beta_{g_1}, \beta_{g_2}, \beta_{g_3}$. After previous steps the attacker has quadratic equations for each β_{s_i}:

$$a\beta_{s_i}^2 + b\beta_{s_i} + c = 0, \tag{10}$$

where $a = \sum_{j=3}^{6} \epsilon_i$, $b = (\epsilon_1 \oplus \epsilon_2)a$ and $c = (\epsilon_1 \oplus \beta_{s_i})(\epsilon_2 \oplus \beta_{s_i})a \oplus (\epsilon_1 \oplus \epsilon_2) \sum_{j=4}^{6} \sum_{k=3}^{j-1} \epsilon_j \epsilon_k \oplus \sum_{j=5}^{6} \sum_{k=4}^{j-1} \sum_{l=3}^{k-1} \epsilon_j \epsilon_k \epsilon_l$ and according to the third condition ϵ_1 and ϵ_2 are elements containing β_{s_i}.

Now the attacker enumerate all the possible initial combinations of values of elements $\beta_{g_1}, \beta_{g_2}, \beta_{g_3}$. Since the attacker looks for linearly independent elements, it holds that

$$\beta_{g_i} \notin span(\{\beta_{g_1}, \dots, \beta_{g_{i-1}}\}). \tag{11}$$

For each combination of values of elements $\beta_{g_1}, \beta_{g_2}, \beta_{g_3}$ roots of these equations are potential candidates for the value of β_{s_i}.

Remaining roots are iterated over to find the solution for β_{s_2} by solving quadratic equations for this element and so on for each β_{s_i}. If the attacker find solutions for all β_{s_i} then whole support is revealed.

Countermeasure

Authors offer two options how to protect this side channel. First countermeasure is similar to the one described in section 1.5.3, thus forcing the XGCD algorithm to iterate more times than needed to find the inverse polynomial.

Second option is to use only $t-1$ errors during the encryption and then to add one more error before decryption. The position of this error should be pseudo-randomly derived from the cipher text and the private key, thus the position of the error for the particular cipher text is always the same. Drawback of this countermeasure is that it is needed to increase security parameters as a compensation for the lower error weight used during the encryption.

2 Timing side-channel attacks against BitPunch implementation

In this section we show how we gained the analyzed data. Furthermore, we present results of attacks against the chosen implementation. For our measurements and attacks we used the BitPunch implementation [1] of the McEliece cryptosystem.

BitPunch is the library implemented in C language by A. Gulyás, M. Klein, J. Kudláč, F. Machovec and F. Uhrecký, without dependencies on external libraries. Our results are based on version 0.0.2 from 1.3.2015. The library is using Goppa codes and is secured by CCA2 schema.

Attacks were realised on the following platform:

- CPU - Intel Core i5-3230M CPU @ 2.60GHz $\times$ 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.

2.1 Measurement methodology

Attacks described in section 1.5 belong to timing side-channel attacks category. For the purpose of measuring execution time, we used the RDTSC instruction [5]. Since tests were realised on the computer with the operating system running many background processes, it is necessary to determine errors in gained data caused by these processes.

Described attacks are aimed at the decryption process, which is deterministic and, therefore, the process should always take the same number of instructions for the same input data. This implies the same time for the same input data. However, empirically gained data disprove this assumption. We assume divergence of sample data is caused by background processes. We can define random variable A describing decryption time for particular input data. We believe random variable A is composed of 2 parts. Specifically, $A = d + E$, where d is constant decryption time and E is variable time of background processes executed during decryption. Purpose of the following test is to find out if the variable A comes from normal distribution. If the variable A comes from normal distribution and d is constant, then E comes from normal distribution. If the variable E comes from normal distribution then the decryption time should be represented as a mean value of obtained data.

Hypothesis
$\mathbf{H}_0$: The variable A comes from normal distribution.
$\mathbf{H}_a$: The variable A does not come from normal distribution.

Decision method
To determine whether data comes from normal distribution we used Kolmogorov-Smirnov test.
Level of significance: 0.05
Sample size: 1000

Test results
$D = 0.2872$
$p - value < 2.2 \times e^{-16}$

Conclusion
Since $p - value < 0.05$ we reject hypothesis $\mathbf{H}_0$.

In the Figure 1 we can see the distribution of measured times of evaluation of the error locator polynomial $\sigma(X)$. Measured values are concentrated at the beginning of the histogram. If we assume the decryption time d constant, then distribution of variable time , represented by the variable E, reflects distribution of the random variable A. This random variable E represents the time consumption of background processes. Since time consumed by background processes can not reduce time of decryption, we assume the shortest measured time is the closest to the actual time of decryption. Thus decryption time for particular input data is the smallest value of the measured data. For better visualization, in Figures 3-24 we present raw timing data from a series of experiments sorted in ascending order.

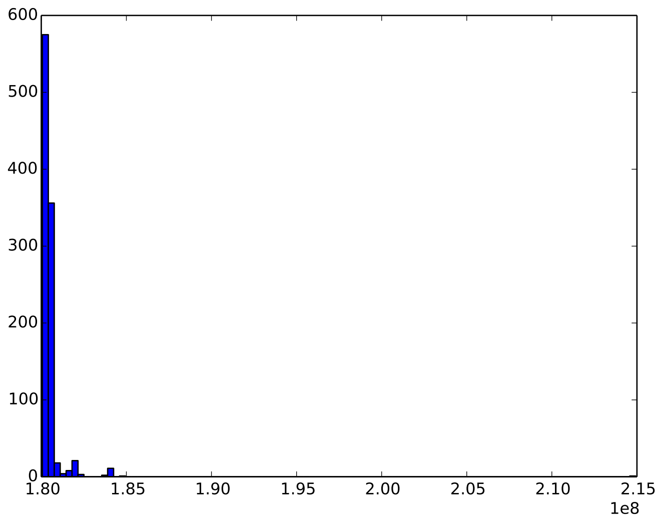

Figure 1: Distribution of empirically gained data.

2.2 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 section 1.5.2.

First we attacked the cryptosystem as in 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 times. With this simple attack we were able to reveal from 45 to 50 errors. Results of the attack are presented in Figure 2.

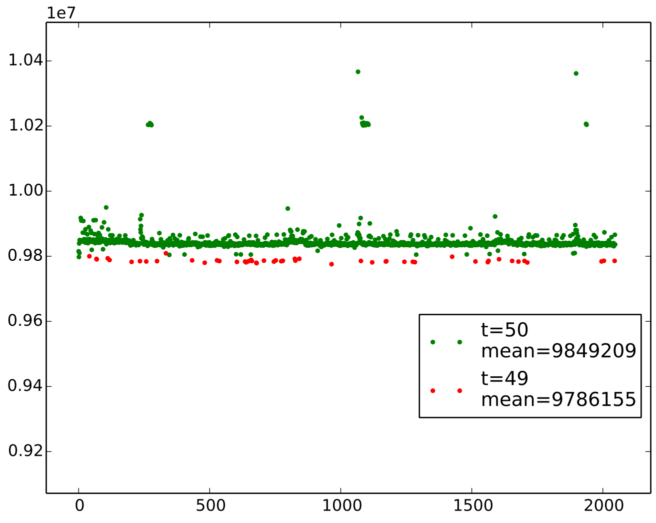

Figure 2: Real-life attack

Decryption times for ciphertexts $\mathbf{c}_i = \mathbf{c} \oplus \mathbf{e}_i$. Red colored points depict error bits.

In Figure 3 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 case when the attacker revealed the bit in the error vector is easily distinguishable from the case when he added the error bit to the ciphertext.

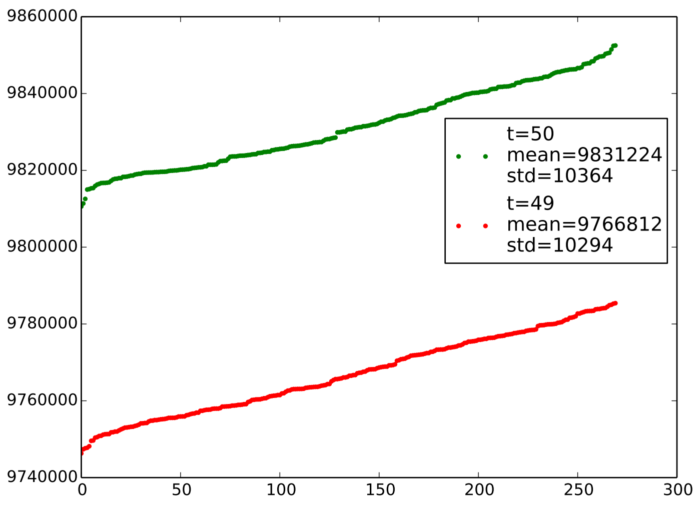

Figure 3: Decryption times

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

2.3 Attack against Patterson Algorithm

In this section we present results of attacking Patterson algorithm. Since time differences between cases $t = 50$ and $t = 48$ are too small to distinguish by measuring the whole decryption process, we present only times needed to find polynomials $a(X)$ and $b(X)$ by the XGCD algorithm.

Despite the fact that the XGCD algorithm needs one less iteration when $t = 48$ than in case of $t = 50$, measured times shows that if attacker reveals positions of error bits, algorithm needs slightly longer time to find $a(X)$ and $b(X)$.

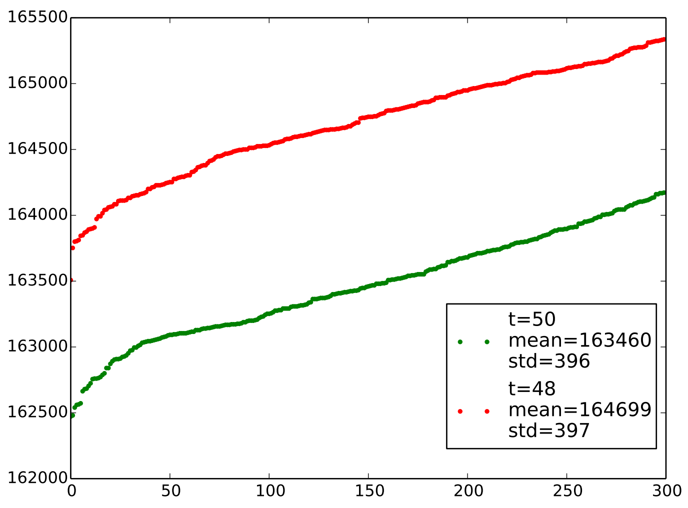

Figure 4: Attack against Patterson algorithm

Times needed to execute the XGCD algorithm.

2.4 Attack against Secret Permutation and Syndrome Inversion

In this section we present results of attacks described in section 1.5.4 and section 1.5.5. However, to point out these time differences we attacked the system with applied countermeasures described in section 3.

These attacks are strongly tied, since they exploit the same vulnerability and time differences caused by revealing $\sigma_3 = 0$ are added one to another. We can see this addition in Table 1. This attack needed on the chosen platform approximately 25 minutes to gain 102 equations like Equation 3, where $\sigma_3(f_1, f_2, f_3, f_4) = 0$.

	Attack against permutation	Attack against inversion	Decryption process
$\sigma_3 \neq 0$	141169	160546	17637240
$\sigma_3 = 0$	60	95770	17416713
difference	141109	64776	220527

Table 1: Attacks against permutation and inversion

3 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 $t = 49$.

3.1 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 5.

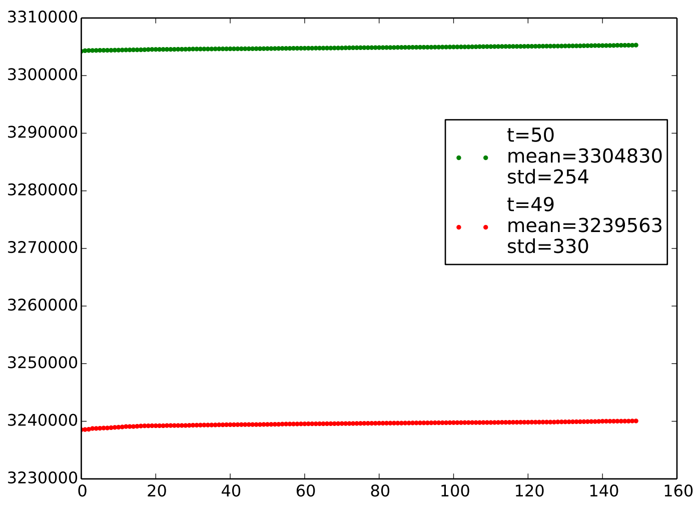

Figure 5: Evaluation of $\sigma(X)$ without countermeasures

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

```c
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) {
  int i;
  BPU_T_GF2_16x ret = 0;
  ret = poly->coef[0];

  for (i = 1; i <= poly->deg; i++) {
    ret = ret ^ BPU_gf2xMulModT(poly->coef[i], BPU_gf2xPowerModT(x, i, math_ctx), math_ctx);
  }
  return ret;
}
```

3.2 Countermeasure against attack on ELP 1

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 which decoding algorithm is capable to correct. The degree of polynomial $\sigma(X)$ is increased at line 2 of Code 3.

Code 3: Countermeasure 1

```c
1 ...
2 sigma.deg = ctx->t;
3 for (l = 0; l < ctx->a_data.ord; l++) {
4     tmp_eval = BPU_gf2xPolyEval(&sigma, ctx->a_data.exp_table[l], &(ctx->a_data));
5     if (tmp_eval == 0) {
6         BPU_gf2VecSetBit(decoded, l, 1);
7     }
8 }
9 ...
```

As we can see in Figure 6 just raising the degree of polynomial $\sigma(X)$ is insufficient. This significant time difference is caused by *"If"* statement at line 5 of Code 3.

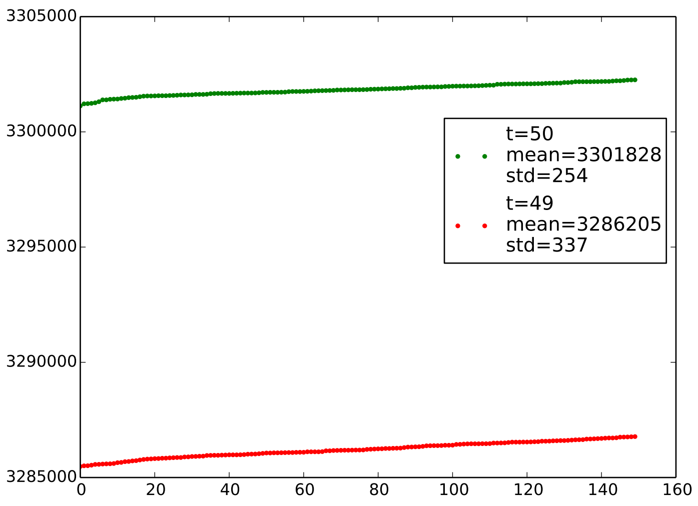

Figure 6: Evaluation of $\sigma(X)$ - countermeasures no. 1

3.3 Countermeasure against attack on ELP 2

By removing the *"If"* statement, mentioned above, time differences between the evaluation of polynomials of degree 50 and 49 were decreased. Nevertheless, both cases are still easily distinguishable.

Code 4: Countermeasure 2

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

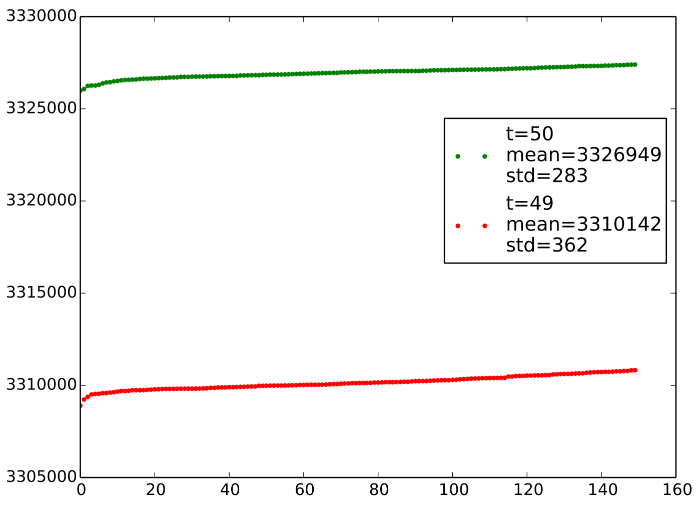

Figure 7: Evaluation of $\sigma(X)$ - countermeasures no. 2

3.4 Countermeasure against attack on ELP 3

Essential part of polynomial evaluation is multiplication of elements in the finite field. This is realised by look-up tables, but as we can see in Code 5 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 at line 3, when the condition is evaluated as true then 0 is returned immediately.

Code 5: Without countermeasure 3

```
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   if (a == 0 || b == 0)
4     return 0;
5   return math_ctx->exp_table[(math_ctx->log_table[a] + math_ctx->log_table[b]) %
        math_ctx->ord];
6 }
7 ...
```

To avoid this difference, it is needed 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 6 at line 5. Result of multiplication is saved in new variable, which is, if needed, returned instead of value computed by look-up tables.

Code 6: Countermeasure 3

```
...
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_32x condition;
  BPU_T_GF2_16x candidate;
  candidate = math_ctx->exp_table[(math_ctx->log_table[a] + math_ctx->log_table[b]) %
      math_ctx->ord];
  if (condition = (a * b))
    return candidate;
  return condition;
}
...
```

In Figure 8 we can see that time differences between evaluation times for polynomials of degree 50 and 49 were decreased, but in Figure 9 we can see that it is 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.

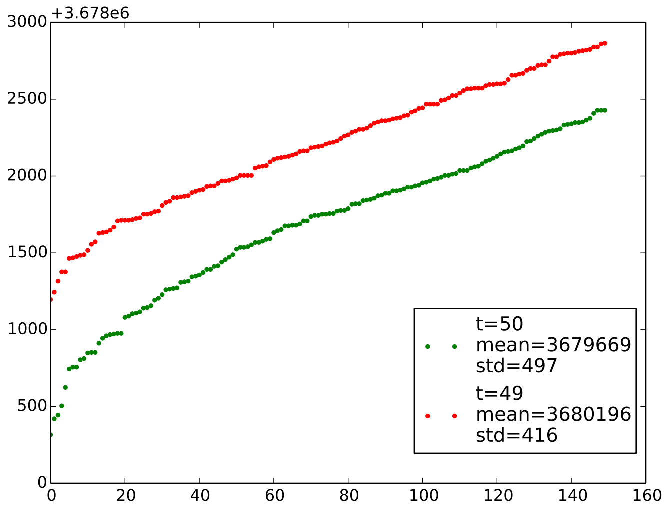

Figure 8: Evaluation of $\sigma(X)$ - countermeasures no. 3

Case when attacker removed one error from error vector.

In Figure 9 we can see time differences between evaluation of polynomial with 50 roots and polynomial with only 1 root, what is the average number of roots in polynomials created by adding one error to the ciphertext.

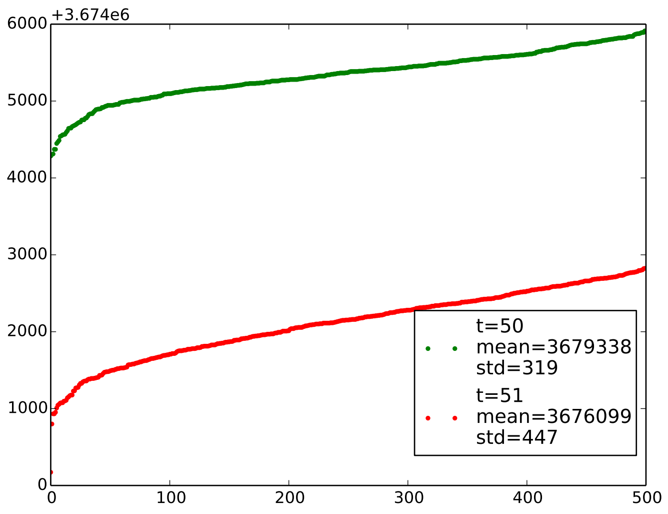

Figure 9: Evaluation of $\sigma(X)$ - countermeasures no. 3

Case when the attacker added one error to the error vector.

3.5 Countermeasure against attack on ELP 4

After the polynomial is evaluated appropriate bit is set to 1 if the result of evaluation is 0, otherwise it is set to 0. This operation is done by macro shown in Code 7. We can see that setting bit to 0 needs one more operation compared to setting bit to 1.

Code 7: Without countermeasure 4

```
#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 8 not only provides the same number of operations but also removes branching which can be used to attack system by power analysis.

Code 8: Countermeasure 4

```
#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);
```

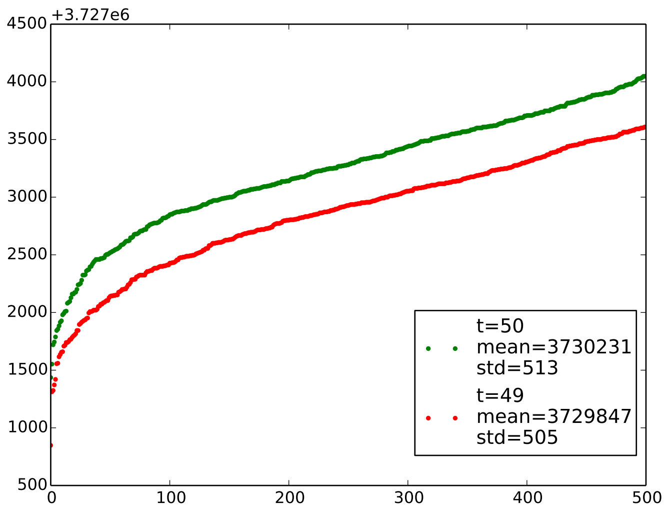

Figure 10: Evaluation of $\sigma(X)$ - countermeasures no. 4

Case when the attacker removed one error from the error vector.

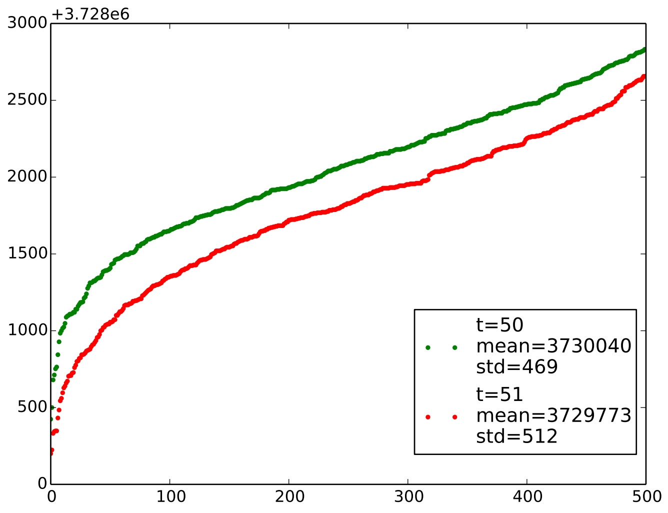

Figure 11: Evaluation of $\sigma(X)$ - countermeasures no. 4

Case when the attacker added one error to the error vector.

3.6 Countermeasure against attack on ELP 5

Another operation used during 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 12, which, as is shown in Code 9, uses only multiplication:

$$\sigma(X) = \sum_{i=0}^{n} a_i X^i = (\dots((a_n X + a_{n-1})X + a_{n-2})X \dots)X + a_1)X + a_0. \tag{12}$$

Code 9: Countermeasure 5

```c
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) {
  int i;
  BPU_T_GF2_16x ret = poly->coef[poly->deg];

  for (i = poly->deg; i > 0; i--) {
    ret = BPU_gf2xMulModT(ret, x, math_ctx) ^ poly->coef[i-1];
  }
  return ret;
}
```

3.7 Countermeasure against attack on ELP 6

After applying 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 10, we can see that the same number of instruction should be used during its execution. Nevertheless, different inputs a and b cause different execution time of this block of code. More specifically, modulo operation in line 5. To highlight this difference, we provide execution times where we increased the number of correctly guessed errors, thus evaluated polynomial $\sigma(X)$ is of lower degree. In Figure 12 we compare execution times for $\sigma(X)$, where $deg(\sigma(X)) = 50$ and $deg(\sigma(X)) = 1$.

Logarithmic and exponential tables are implemented in a following way:
$E[i] = \alpha^i$, where $i = 0, \dots, 2^m - 2$ and $E[2^m - 1] = 0$.
$L[E[i]] = i$, where $i = 0, \dots, 2^m - 2$ 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 $L[a] + L[B] >= D$, thus zero coefficients of $\sigma(X)$ are causing this time difference.

Code 10: Without countermeasure 6

```c
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    candidate = math_ctx->exp_table[(math_ctx->log_table[a] + math_ctx->log_table[b]) % math_ctx->ord];
6    if (condition = (a * b))
7      return candidate;
8    return condition;
9  }
10 ...
```

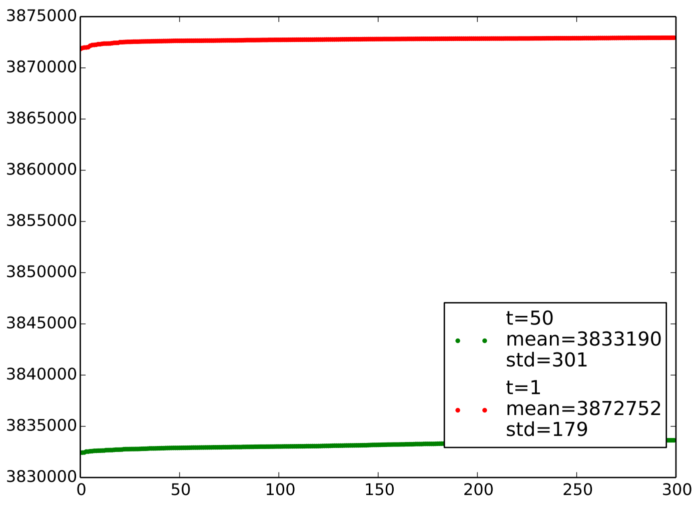

Figure 12: Evaluation of $\sigma(X)$ - standard modulo operation

In Code 11 modulo operation is replaced by the code at lines $6-10$. This replacement of modulo operation is not only time constant but also faster than previous version. Unfortunately this countermeasure is not possible to apply on cryptosystem where parameter $n \neq 2^m$.

Code 11: With countermeasure 6

```
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_16x candidate;
4    BPU_T_GF2_16x exp, bit, carry_mask = 1 << math_ctx->mod_deg;
5    BPU_T_GF2_32x condition;
6    exp = math_ctx->log_table[a] + math_ctx->log_table[b];
7    exp = exp + 1;
8    bit = (exp & carry_mask);
9    exp = (exp & math_ctx->ord);
10   exp = (exp & math_ctx->ord) - !bit;
11   candidate = math_ctx->exp_table[exp];
12   if (condition = (a * b))
13     return candidate;
14   return condition;
15 }
16 ...
```

In Figure 13 we can see that measured times for polynomials $\sigma(X)$ of degrees 50 and 49 are approximately the same. Nevertheless, multiple measurements, shown in Table 2, have shown that difference between average times can vary, but in a way that it is not possible to distinguish between guessing the correct position, or wrong position of error bit. It means that times of evaluation for modified ciphertexts oscillate around the time of evaluation for unmodified ciphertexts.

	T_1	T_2	T_3	T_4	T_5	T_6
$deg(\sigma(X)) = 50$	2411695	2412237	2412051	2426570	2412048	2412099
$deg(\sigma(X)) = 49$	2411884	2412075	2411986	2426657	2411929	2412113
Difference	-189	162	65	-87	119	-14

Table 2: table

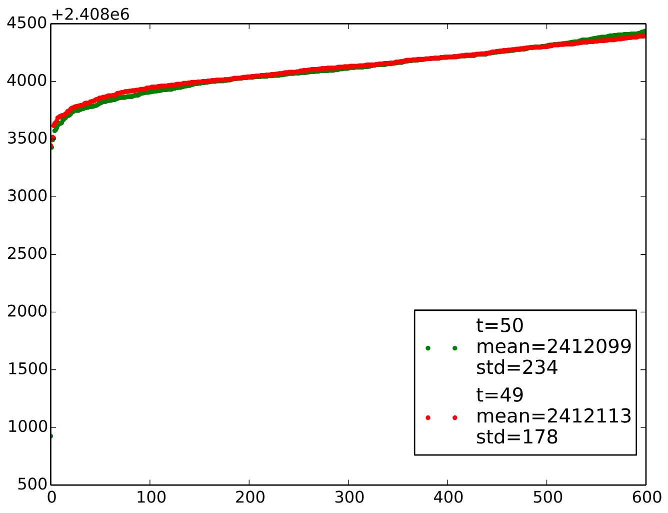

Figure 13: Evaluation of $\sigma(X)$ - countermeasure no. 6

However, in Figure 14 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.

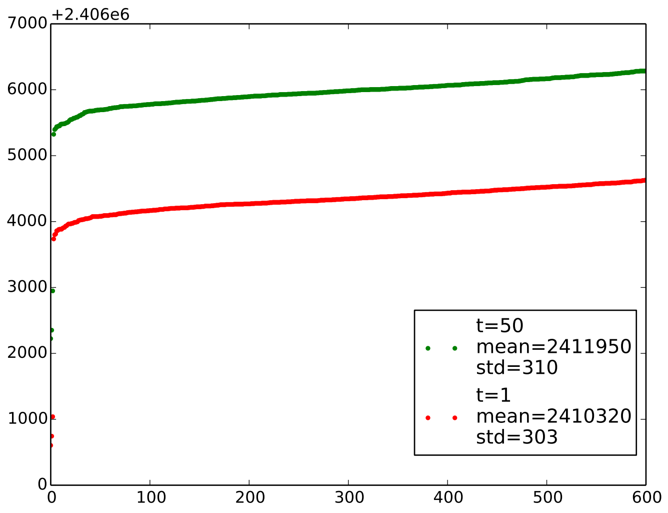

Figure 14: Evaluation of $\sigma(X)$ - countermeasure no. 6

3.8 Countermeasure against attack on ELP 7

Since frequently used data can be stored in the CPU cache for faster access of processor to data, time differences pointed out in Figure 14 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 12. Unfortunately, this multiplication is approximately 2.5 times slower.

Code 12: 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 15 we can see that evaluation times for polynomials $\sigma(X)$ of degrees 50 and 49 are approximately the same, but as mentioned above, they are not exactly the same but oscillate around same values, see Table 3.

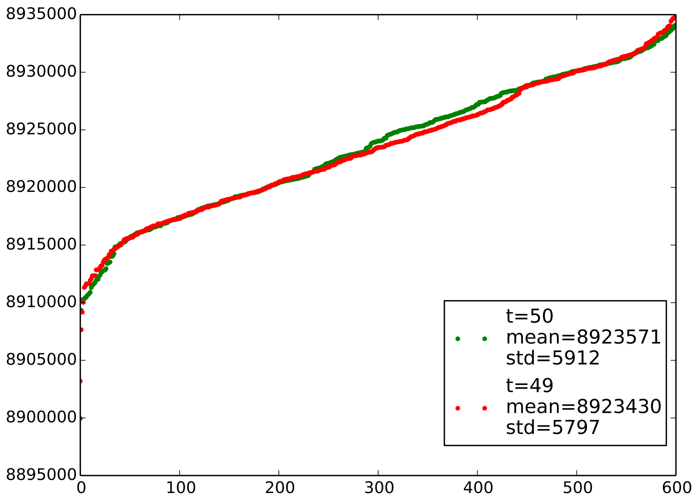

Figure 15: Evaluation of $\sigma(X)$ - countermeasure no. 7

	T_1	T_2	T_3	T_4	T_5	T_6
$deg(\sigma(X)) = 50$	8923666	8921522	8919976	8919626	8919285	8923571
$deg(\sigma(X)) = 49$	8924178	8920578	8919431	8920110	8919548	8923430
Difference	-512	944	545	-484	-263	141

Table 3: Evaluation times of $\sigma(X)$ of degree 50 and 49

Similar results were achieved when evaluation of polynomial $\sigma(X)$ of degree 50 was compared to evaluation of polynomial $\sigma(X)$ of degree 1, see Figure 16 and Table 4.

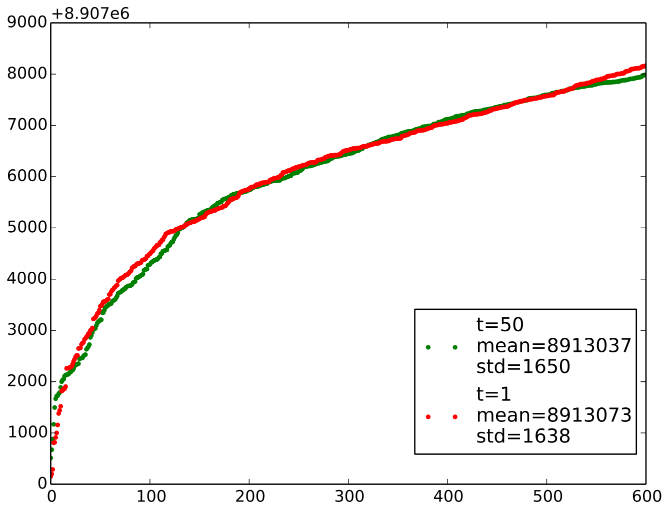

Figure 16: Evaluation of $\sigma(X)$ - countermeasure no. 7

	T_1	T_2	T_3	T_4	T_5	T_6
$deg(\sigma(X)) = 50$	8916516	8918506	8912341	8912715	8912524	8913037
$deg(\sigma(X)) = 1$	8917032	8919096	8911855	8912428	8912581	8913073
Difference	-516	-590	486	287	-57	-36

Table 4: Evaluation times of $\sigma(X)$ of degree 50 and 1

3.9 Conclusion

Proposed countermeasures should avoid the attack described in section 1.5.2 in a way that 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 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.

4 Countermeasures against XGCD based attacks

In the following we show the code which is vulnerable to timing attacks described in section 1.5.3, section 1.5.4 and section 1.5.5. As we can see, the number of iterations of this algorithm depends on degrees of polynomials $old\_r(X)$ and $r(X)$, where $old\_r(X)$ corresponds to the polynomial $g(X)$ and $r(X)$ corresponds to $\tau(X)$ from alg. 8.

Code 13: XGCD - naive implementation

```
while (old_r.deg > end_deg && r.deg > -1) {
  BPU_gf2xPolyDiv(&q, &tmp, &old_r, &r, math_ctx); // oldr_r : r = q + tmp

  // save old reminder
  BPU_gf2xPolyCopy(&old_r, &r); // old_r := r
  // save current reminder
  BPU_gf2xPolyCopy(&r, &tmp); // r := tmp

  // save s quocient
  // (old_s, s) := (s, old_s + qs)
  BPU_gf2xPolyCopy(&tmp, &old_s);
  BPU_gf2xPolyCopy(&old_s, s);
  BPU_gf2xPolyMul(&tmp_2, &q, s, math_ctx);
  BPU_gf2xPolyAdd(s, &tmp, &tmp_2);

  // save t quocient
  // (old_t, t) := (t, old_t + qt)
  BPU_gf2xPolyCopy(&tmp, &old_t);
  BPU_gf2xPolyCopy(&old_t, t);
  BPU_gf2xPolyMul(&tmp_2, &q, t, math_ctx);
  BPU_gf2xPolyAdd(t, &tmp, &tmp_2);
}
```

4.1 Observation

Despite the fact that number of iterations in Code 13 decreases with the number of errors in ciphertext, in Figure 17, Figure 18 and Figure 19 we can see it is not directly proportional.

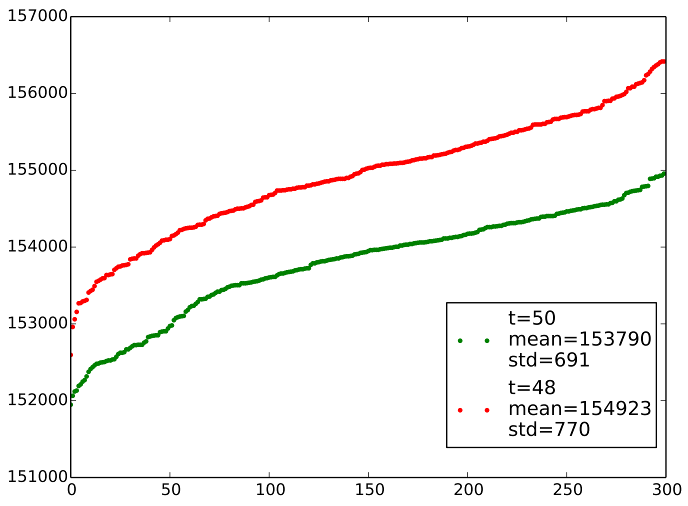

Figure 17: XGCD - 25 iterations compared to 24 iterations

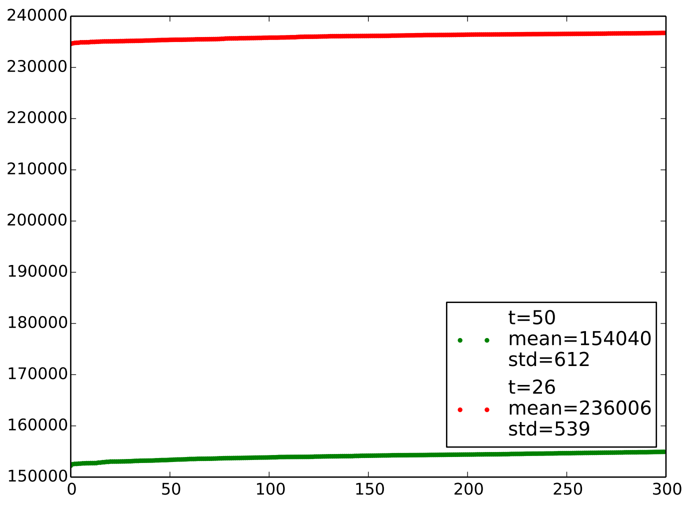

Figure 18: XGCD - 25 iterations compared to 13 iterations

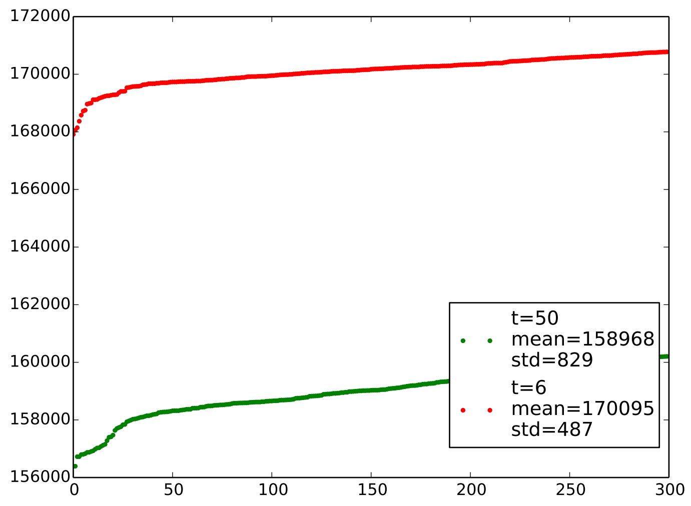

Figure 19: XGCD - 25 iterations compared to 3 iterations

This disproportion is caused by the number of iterations in XGCD in combination with the number of iterations in polynomial division, executed at line 2 of Code 13.

When we look at the implementation of polynomial division, listed in Code 14, we can see that the number of iterations is $i = deg(a(X)) - deg(b(X)) + 1$. Furthermore, at line 19, polynomial multiplication is executed and its complexity depends on degrees of polynomial $q(X)$ and $divider(X)$, where $divider(X)$ is just denoted polynomial $b(X)$. Since the complexity of multiplication of two polynomials of degrees b and q is $\mathcal{O}((b+1)(q+1))$, we can denote the complexity of polynomial division as $\mathcal{O}((a-b+1)(q+1)(b+1))$, where a, b and q are degrees of mentioned polynomials. Since $q = a - b$ we can rewrite the complexity as $\mathcal{O}((a-b+1)(a-b+1)(b+1)) = \mathcal{O}((a-b+1)^2(b+1))$. For better understanding how these degrees influence the execution time of XGCD algorithm see Table 5.

	$t = 50$			$t = 26$			$t = 6$		
iter. no.:	a	b	Complexity	a	b	Complexity	a	b	Complexity
1	50	49	$\mathcal{O}(4 \times 50)$	50	49	$\mathcal{O}(4 \times 50)$	50	49	$\mathcal{O}(4 \times 50)$
2	49	48	$\mathcal{O}(4 \times 49)$	49	48	$\mathcal{O}(4 \times 49)$	49	48	$\mathcal{O}(4 \times 49)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	48	3	$\mathcal{O}(2116 \times 4)$
12	39	38	$\mathcal{O}(4 \times 39)$	39	38	$\mathcal{O}(4 \times 39)$	-	-	-
13	38	37	$\mathcal{O}(4 \times 38)$	38	13	$\mathcal{O}(676 \times 14)$	-	-	-
$\vdots$	$\vdots$	$\vdots$	$\vdots$	-	-	-	-	-	-
24	27	26	$\mathcal{O}(4 \times 27)$	-	-	-	-	-	-
25	26	25	$\mathcal{O}(4 \times 26)$	-	-	-	-	-	-
Summary			$\mathcal{O}(3800)$			$\mathcal{O}(11600)$			$\mathcal{O}(8860)$

Table 5: Complexity of XGCD when $t = 50$, $t = 26$ and $t = 6$

Note that a and b corresponds to input parameters of function BPU_gf2xPolyDiv, thus $r_{i-2}(X)$ and $r_{i-1}(X)$ from alg. 8 and not to polynomials $a(X)$ and $b(X)$ which are results of Patterson algorithm.

Code 14: Polynomial division

```c
void BPU_gf2xPolyDiv(BPU_T_GF2_16x_Poly *q, BPU_T_GF2_16x_Poly *r, const
    BPU_T_GF2_16x_Poly *a, const BPU_T_GF2_16x_Poly *b, const BPU_T_Math_Ctx *math_ctx) {
  ...
  // a:b = q+r
  BPU_gf2xPolyCopy(&dividend, a);
  *divider = b
  for (i = a->deg; i >= 0; i--) {
    if (dividend.deg < divider->deg) {
      BPU_gf2xPolyCopy(r, &dividend);
      break;
    }
    BPU_gf2xPolyNull(&tmp);
    leader = BPU_gf2xMulModT(dividend.coef[i],
        BPU_gf2xPowerModT(divider->coef[divider->deg], -1, math_ctx), math_ctx);
    exponent = dividend.deg - divider->deg;
    q->coef[exponent] = leader;

    if(q->deg == -1) {
      q->deg = BPU_gf2xPolyGetDeg(q);
    }
    BPU_gf2xPolyMul(&tmp, divider, q, math_ctx);
    BPU_gf2xPolyAdd(&dividend, a, &tmp);
  }
  ...
}
```

4.2 Countermeasure against the attack on Patterson algorithm 1

In order to achieve the constant running time of the XGCD algorithm, first we need to force it to run constant number of iterations. We decided to control the degree of polynomial $old\_r(X)$. Hence, we lower the degree of this polynomial by one in each iteration. Moreover, we avoided multiplication of polynomials $q(X)s(X)$ and $q(X)t(X)$, executed at lines 13 and 20 in Code 13, by continuous multiplication of the polynomial $t(X)$ by the coefficient q_i followed by shifting this polynomial according to the power of X at the corresponding coefficient. We can omit this step for the polynomial $s(X)$ since $s(X)$ is not needed and instead of that we use this polynomial to perform "dummy" operations. This "dummy" operations serve to mask the process of saving polynomial $t(X)$ and swapping polynomials $old\_r(X)$ and $r(X)$.

Note that for multiplication of the polynomial $t(X)$ by the coefficient q_i we used countermeasure discussed in section 3.7. Countermeasure discussed in section 3.8 yields similar results but is significantly slower.

Code 15: XGCD with countermeasure

```
parity = (!(end_deg & 1)) * 2;
for (counter = 0; counter < end_deg*2 + parity; counter++) {
  // leader = old_r.coef[act_deg] / r.coef[r.deg];
  leader_exp = math_ctx->log_table[old_r.coef[act_deg]] -
      math_ctx->log_table[r.coef[r.deg]];
  if (leader_exp < 0) {
    leader_exp += math_ctx->ord;
  }
  leader = math_ctx->exp_table[leader_exp];
  leader = leader * (!!old_r.coef[act_deg]);
  // old_r = old_r + r tmp_q(X);
  BPU_gf2xPolyCopy(&helper1, &r);
  BPU_gf2xPolyMulEl(&helper1, leader, math_ctx);
  BPU_gf2xPolyShlC(&helper1, act_deg - r.deg);
  BPU_gf2xPolyAddC(&helper2, &old_r, &helper1);
  BPU_gf2xPolyCopy(&old_r, &helper2);
  // q = q + tmp_q(X)
  BPU_gf2xPolyCopy(&helper1, t);
  BPU_gf2xPolyMulEl(&helper1, leader, math_ctx);
  BPU_gf2xPolyShlC(&helper1, act_deg - r.deg);
  BPU_gf2xPolyCopy(&helper2, &tmp_2);
  BPU_gf2xPolyAddC(&tmp_2, &helper1, &helper2);
  // prepare swap(old_r, r)
  BPU_gf2xPolyCopy(&helper1, &old_r);
  BPU_gf2xPolyCopy(&old_r, &r);
  if (r.deg < act_deg) {
    act_deg -= 2;
    // cancel swap(old_r, r)
    BPU_gf2xPolyCopy(&old_r, &helper1);
    // dummy process
    BPU_gf2xPolyCopy(&tmp, &old_s);
    BPU_gf2xPolyCopy(&old_s, s);
    BPU_gf2xPolyAddC(s, &tmp, &tmp_2);
    BPU_gf2xPolyNull(&tmp);
  }
  else {
    act_deg -= 1;
    // finish swap(old_r, r)
    BPU_gf2xPolyCopy(&r, &helper1);
    // save t quocient
    BPU_gf2xPolyCopy(&tmp, &old_t);
    BPU_gf2xPolyCopy(&old_t, t);
    BPU_gf2xPolyAddC(t, &tmp, &tmp_2);
    BPU_gf2xPolyNull(&tmp_2);
  }
  act_deg += 1;
}
```

In Figure 20 we can see times needed to find polynomials $a(X)$ and $b(X)$ for cases of number of errors is $t = 50$ and $t = 48$. We can consider these times indistinguishable. However, in Figure 21 we can see it is still possible to distinguish case $t = 50$ from case $t = 4$.

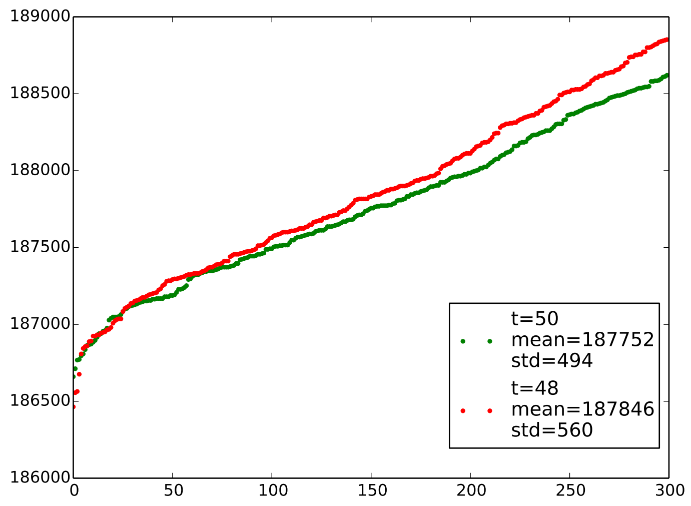

Figure 20: XGCD - execution times for ciphertexts containing 50 and 48 errors

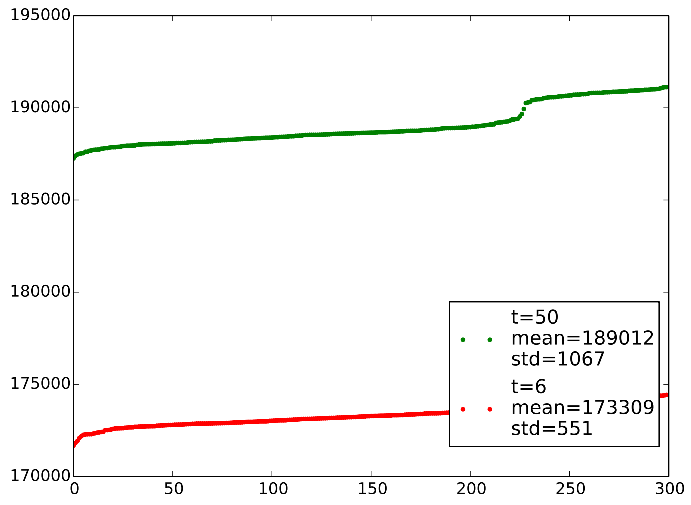

Figure 21: XGCD - execution times for ciphertexts containing 50 and 6 errors

4.3 Countermeasure against attack on Patterson algorithm 2

Time differences pointed out in Figure 21 are partially caused by function BPU_gf2xPolyGetDeg, which is called at the end of function BPU_gf2xPolyAddC. This function is implemented as in Code 16. This implementation does not have constant execution time, since it is terminated when first non-zero coefficient is found.

We protected this function by iterating through all coefficients and saving the position of first non-zero coefficient, for detailed implementation see Code 17.

This countermeasure reduced time differences for the ciphertext containing 6 errors but it is still distinguishable. These differences might be caused by operating system accessing different part of the memory while executing "dummy" process.

Code 16: Get degree of polynomial without countermeasure

```c
int BPU_gf2xPolyGetDeg(BPU_T_GF2_16x_Poly *poly) {
  int i = poly->max_deg;
  for (i; i >= 0; i--) {
    if (poly->coef[i] != 0) {
      return i;
    }
  }
  return i;
}
```

Code 17: Get degree of polynomial with countermeasure

```c
int BPU_gf2xPolyGetDegC(BPU_T_GF2_16x_Poly *poly) {
  int i = poly->max_deg;
  int deg = 0;
  for (i; i >= 0; i--) {
    deg = deg ^ ((i + 1) * !deg * !!poly->coef[i]);
  }
  deg = deg - 1;
  return deg;
}
```

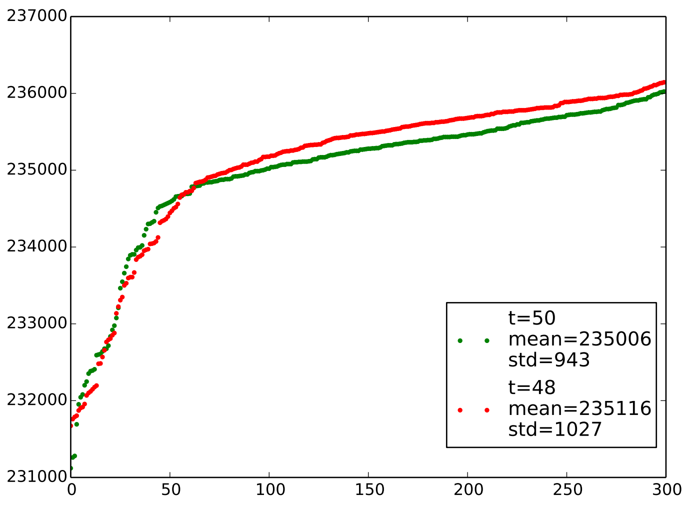

Figure 22: XGCD - execution times for ciphertexts containing 50 and 48 errors

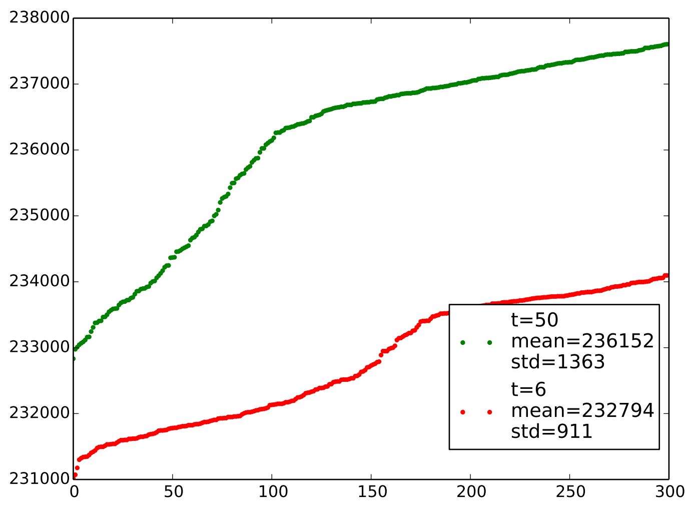

Figure 23: XGCD - execution times for ciphertexts containing 50 and 6 errors

4.4 Conclusion

Described countermeasures should be sufficient to avoid timing attacks described in section 1.5.3 and section 1.5.4. However, the implementation is still vulnerable to the attack described in section 1.5.5 but this can be avoided by changing encryption schema, specifically, by adding one more error to the ciphertext before its decryption.

5 Conclusions

In this thesis we summarized known timing attacks on the McEliece cryptosystem. We realised chosen timing side-channel attacks against BitPunch implementation and we pointed out its vulnerabilities. In order to obtain more precise measurements we improved measurement methodology. We have implemented countermeasures against chosen timing attacks and evaluated their efficiency. We pointed out the necessity of considering not only timing attacks caused by insecure software implementation, but also cache timing attacks which depend on chosen hardware platform. Moreover, we improved XGCD algorithm.

We avoided the most significant attack, the attack on the error locator polynomial. Results of our countermeasures are presented in Figure 24 and Table 6.

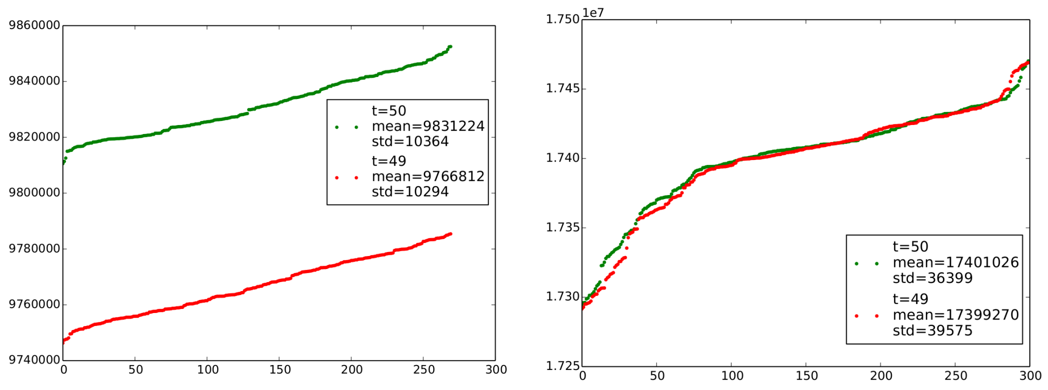

Figure 24: Comparison of naive implementation and secured implementation.

	T_1	T_2	T_3	T_4	T_5	T_6
$deg(\sigma(X)) = 50$	17368225	17379647	17405700	17382657	17378325	17401026
$deg(\sigma(X)) = 49$	17370285	17376275	17408089	17379749	17381780	17399270
Difference	-2060	3372	-2389	2908	-3455	1756

Table 6: Decryption times of secured code for modified and non-modified ciphertexts.

Since these countermeasures significantly slow down the decryption process, further research should be focused on effective implementation of these countermeasures.

The slowest countermeasure is the multiplication in a finite field. This operation can be easily implemented in a hardware, thus 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. Furthermore, improvements in the XGCD algorithm are needed, since vulnerability described in section 1.5.5 has not been secured yet.

This attacks and measurements were realised on computer running many background processes, thus we recommend to repeat evaluate efficiency of countermeasures on embedded devices. Furthermore, program executed in constant time can be used as a base for simple and differential power analysis attacks.

Moreover, further research is needed, since this thesis is aimed at attacks realised during decryption but there exists attacks possible to realise during the key generations e.g., attack on generation of parity check matrix **H** presented in [11].

Résumé

Cieľom diplomovej práce je analýza bezpečnosti SW implementácie McElieceovho kryptosystému. Diplomová práca je zameraná na odolnosť BitPunch implementácie McElieceovho kryptosystému voči útokom postrannými kanálmi, konkrétne voči časovým útokom.

Jedným z hlavných cieľov práce je poukázať na fakt, že “naivná” implementácia kryptosystému môže mať fatálne následky na jeho reálnu bezpečnosť. Ďalším cieľom práce je zabezpečiť knižnicu BitPunch proti zvoleným útokom.

Práca sa skladá z piatich častí. V časti 1 Analysis sa nachádza stručný úvod do teórie kódovania, McElieceovho kryptosystému a detailný popis analyzovaných útokov.

V časti 2 Timing side-channel attacks against BitPunch implementation je popísaná metodika merania a výsledky realizovaných útokov. Táto časť poukazuje na zraniteľnosť zvolenej implementácie.

V časti 3 Countermeasures against ELP based attack a 4 Countermeasures against XGCD based attacks poukazujeme na chyby v implementácii a prezentujeme zvolené protiopatrenia a ich účinnosť.

V časti 5 Conclusions sa nachádza zhodnotenie výsledkov práce, zhodnotenie účinnosti zvolených protiopatrení a problémov s nimi spojenými a navrhujeme ciele ďalšieho výskumu.

Keďže zvolené protiopatrenia značne spomalili proces dešifrovania, ďalší výskum by sa mal zaoberať ich efektívnou implementáciou. Navyše, takto zabezpečená implementácia môže slúžiť na realizáciu SPA a DPA útokov.

References

[1] Gulyás A., Klein M., Kudláč J., Machovec F., and Uhrecký F. Reálna implementácia code-based cryptography. 2014.

[2] Cameron F. Kerry, Acting Secretary, and Charles R. Director. FIPS PUB 186-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS). 2013.

[3] Kazukuni Kobara and Hideki Imai. Semantically secure mceliece public-key cryptosystems -conversions for mceliece pkc -. In Kwangjo Kim, editor, *Public Key Cryptography*, volume 1992 of *Lecture Notes in Computer Science*, pages 19–35. Springer Berlin Heidelberg, 2001.

[4] Robert J McEliece. A public-key cryptosystem based on algebraic coding theory. *DSN progress report*, 42(44):114–116, 1978.

[5] Gabriele Paoloni. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures.* Intel Corporation.

[6] N. Patterson. The algebraic decoding of Goppa codes. *IEEE Transactions on Information Theory*, 21(2):203–207, March 1975.

[7] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[8] Abdulhadi Shoufan, Falko Strenzke, H. Gregor Molter, and Marc Stöttinger. A timing attack against patterson algorithm in the mceliece PKC. In Donghoon Lee and Seokhie Hong, editors, *Information, Security and Cryptology - ICISC 2009, 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers*, volume 5984 of *Lecture Notes in Computer Science*, pages 161–175. Springer, 2009.

[9] Falko Strenzke. A timing attack against the secret permutation in the mceliece PKC. In Nicolas Sendrier, editor, *Post-Quantum Cryptography, Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25-28, 2010. Proceedings*, volume 6061 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 2010.

[10] Falko Strenzke. Timing attacks against the syndrome inversion in code-based cryptosystems. In Philippe Gaborit, editor, *Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings*, volume 7932 of *Lecture Notes in Computer Science*, pages 217–230. Springer, 2013.

[11] Falko Strenzke, Erik Tews, H. Gregor Molter, Raphael Overbeck, and Abdulhadi Shoufan. Side channels in the mceliece PKC. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, volume 5299 of *Lecture Notes in Computer Science*, pages 216–229. Springer, 2008.

Appendices

A Contents of enclosed CD-rom

The enclosed CD-rom contains following directories:

- source
- thesis
- data
  - basic_attack
  - decryption_counter
  - elp
  - patterson
  - patterson_inversion

The “source” directory contains source code of BitPunch implementation of the McEliece PKC with countermeasures.
The “data” directory contains measurements and graphs for attacks and countermeasures presented in this thesis. For detailed explanation see README.txt files.
The “thesis” directory contains this thesis in pdf format and latex source files.

B Algorithms

Algorithm 8 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 $deg(a) \leq d_{break}$
 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 $b(X)$
```

Algorithm 9 Patterson Algorithm

```
Require: $n$-bit word $\mathbf{c}$, Goppa polynomial $g(X)$.
Ensure: $n$-bit error vector $\mathbf{e}$.
 1: Compute syndrome polynomial $S_{\mathbf{c}}(X) = \mathbf{c}\mathbf{H}^T \left(X^{t-1}, \ldots, X, 1\right)^T$, where $\mathbf{H}$ 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 $deg(a) \leq \lfloor \frac{t}{2} \rfloor$.
 5: Determine error locator polynomial $\sigma(X) = a^2(X) + xb^2(X)$, where $deg(\sigma) \leq t$.
 6: Reconstruct the error vector $\mathbf{e} = (\sigma(\alpha_0), \ldots, \sigma(\alpha_{n-1})) \oplus (1, \ldots, 1)$.
 7: return $\mathbf{e}$
```

Algorithm 10 Polynomial Division

```
Require: $n(X), d(X)$
Ensure: $q(X), r(X)$ such that $q(X)d(X) + r(X) = n(X)$.
 1: $r_0(X) = n(X)$
 2: $q_0 = 0$
 3: $i = 0$
 4: while $deg(r_i(X)) \geq deg(d(X))$ do
 5:     $i = i + 1$
 6:     $a_i = r_{i-1,deg(r_{i-1}(X))}/d_{deg(d(X))}$
 7:     $f_i = deg(r_{i-1}(X)) - deg(d(X))$
 8:     $q_i(X) = q_{i-1}(X) + a_i X^{f_i}$
 9:     $r_i = r_{i-1}(X) + a_i X^{f_i} d(X)$
10: return $q_i(X)$ and $r_i(X)$
```