UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Douglas Marcelino Beppler Martins

Root finding techniques in binary finite fields

Florianópolis
2019

Douglas Marcelino Beppler Martins

Root finding techniques in binary finite fields

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do título de Mestre em Ciência da Computação.
Orientador: Prof. Ricardo Felipe Custódio, Dr.

Florianópolis
2019

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Beppler Martins, Douglas Marcelino
Root finding techniques in binary finite fields / Douglas Marcelino Beppler Martins ; orientador, Ricardo Felipe Custódio, 2020.
70 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2020.

Inclui referências.

1. Ciência da Computação. 2. Side-channel Attack. 3. Post-quantum Cryptography. 4. Code-based Cryptography. 5. Root Finding. I. Custódio, Ricardo Felipe. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

Douglas Marcelino Beppler Martins
Root finding techniques in binary finite fields

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Daniel Panario, Dr.
Carleton University

Prof. Jean Everson Martina, Dr.
Universidade Federal de Santa Catarina

Prof. Martín Augusto Gagliotti Vigil, Dr.
Universidade Federal de Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Mestre em Ciência da Computação.

Prof. Vania Bogorny, Dr
Coordenadora do Programa

Prof. Ricardo Felipe Custódio, Dr.
Orientador

Florianópolis, 2019.

To my grandfather, Marcelino Vieira Filho.

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere gratitude to my Family. To my mom, Maria Goreti, thank you for the support that you gave for me always. I wish to thank my father, Marcio Luiz Martins, for teaching me values that I will carry for all my life. My sister, Tais, thank you for helping me to take care of our family. Last but not least, thank you Tamiris, for being one of my academic inspirations and for giving me the greatest gift that I received during my master studies, our lovely Miguel.

I am profoundly grateful to my life partner, Julia de Assis da Cunha. Thanks for being patient with me when I was stressed, for encouraging me when I was afraid. Thanks for listening to me presenting the LATINCRYPT paper, even without understanding anything. Thanks for always supporting me on my choices and for staying by my side. I love you.

I wish to say thanks to my friends from my research group, Gustavo Zambonin for the partnership during all my master studies and all reviews during this journey. To Douglas Silva, Matheus Bittencourt, Ramna Siddhartha for all relaxed conversations during the working time. Also, I extend my gratitude to all colleagues from LabSEC. I hope to take your friendships for all my life.

I wish to express my deepest gratitude to my childhood friends, Ronan, Lucas, and Vitor. The distance does not represent the gratitude that I have for being your friend. And thank you Ramon, Mauricio, José Araújo, and your respective partners, for all Cuba-libre that we drink in the last years, this truly helps me to forget my academic problems during the more stressful moments.

In the end, I wish to say thank you to those who directly helped me to construct this master thesis. Thanks Gustavo Banegas for presenting me the idea of the attack, the root-finding problem in code-based and for being my coauthor. Thanks Daniel Panario for all help on the code-based background and in root-finding methods, for all reviews and all ‘cervejinhas’ during this journey. Also, thanks to Jean Martina and Martin Vigil for the committee review that helped to enrich the final version of this work. I would like to thank my supervisor Ricardo Custodio, thanks for accepting to guide me and for being patient with my routine to always finish my tasks really close to the deadlines.

I dedicate this thesis to all of you.

This is not the end. It is not even the beginning of the end.
But it is, perhaps, the end of the beginning.
Winston Churchill

RESUMO

Com a disseminação da Internet, a comunicação entre pessoas e o acesso a informação se tornou instantâneo. O principal mecanismo que garante essa comunicação de forma segura e privada é a criptografia. Atualmente, os protocolos criptográficos utilizados para realizar a troca de mensagens segura são baseados em problemas matemáticos que podem ser considerados inseguros com o avanço da computação quântica. A fim de criar novos padrões de criptografia, o Instituto de Padrões e Tecnologia (NIST), abriu um processo de padronização de esquemas que sejam seguros contra ataques clássicos e quânticos. Como consequência, a implementação desses novos esquemas precisa ser realizada de forma com que não seja possível realizar um ataque de efeito colateral. Este trabalho apresenta um ataque a um esquema criptográfico baseado em códigos de correção de erro submetido ao NIST, o BIGQUAKE. Este esquema é baseado no clássico trabalho de McEliece, e em sua fase de decodificação, durante o algoritmo de Paterson, os autores do BIGQUAKE não utilizam um método constante. Adicionalmente, contramedidas a esse ataque são apresentadas, comparadas e avaliadas para que seja possível construir um esquema seguro. **Palavras-chave:** Ataque de efeito colateral. Criptografia pós-quantica. Criptografia

baseada em códigos de correção de erro. Raizes de polinomios.

RESUMO ESTENDIDO

Introdução

A segurança dos algoritmos de criptografia assimétrica depende de problemas computacionalmente difíceis, os quais o tempo para um computador conseguir quebrar o esquema deve ser suficientemente grande, a ponto de se tornar inviável. Esquemas atuais tem sua segurança baseada na fatoração de um número composto por dois primos grandes(1024 ou 2048 bits). Outro problema comumente utilizado é o logaritmo discreto. Estes dois problemas computacionais, atualmente, ainda são considerados seguros. Entretanto, com o avanço dos computadores quânticos nos últimos anos, e o conhecido algoritmo de Shor fazem com precisamos encontrar novos esquemas criptográficos que sejam resistentes a ataques quânticos. Uma classe principais candidatos a substituto são esquemas baseados em códigos de correção de erro, mais especificamente, os baseados no criptossistema de McEliece.
O criptossistema de McEliece é considerado seguro tanto quanto ataques quânticos quanto como ataques feitos em computadores atuais. Porém, sua implementação está sujeita a ataques de canal lateral em sua decodificação. Este trabalho propõe explorar tal vulnerabilidade no criptosistema chamado BIGQUAKE, que foi submetido ao *National Institute of Standards and Technology*. Além disso, é proposto 5 contramedidas para este tipo de ataque, de forma que seja possível realizar a implementação do criptossistema de McEliece de forma segura.

Objetivos

O principal objetivo deste trabalho é o estudo e o desenvolvimento de métodos seguros para realizar a decodificação em criptossistemas baseados em códigos de correção de erros. Evitando ataques de canal lateral baseados no tempo de execução do algoritmo. Mais especificamente, este trabalho busca métodos para encontrar raízes de polinômios sobre extensões de corpos finitos.

Metodologia

Com o objetivo de alcançar os objetivos deste trabalho, foi adotado uma metodologia de pesquisa, a qual após o amplo estudo de todos os conceitos básicos da área de pesquisa, se inicio uma revisão da literatura. Pesquisando nas principais ferramentas de busca científica, artigos e trabalhos acadêmicos sobre ataques de canal lateral em criptossistemas baseados em códigos. Foram considerados os trabalhos mais relevantes encontrados nos principais artigos, tanto de conferências quanto de periódicos. Após o levantamento dos principais trabalhos, foram comparados as principais estratégias encontradas para proteger um criptossistema. A partir dessa comparação, novas estratégias foram propostas, implementadas, analisadas e comparadas.

Resultados e Discussão

Este trabalho tem como resultado inicial a execução de um ataque de canal lateral baseado na variação do tempo em uma proposta de esquema resistente a ataques quânticos submetido ao NIST. Com um computador comum, em dezessete minutos é possível recuperar uma mensagem em claro, que no protocolo em questão é uma chave de sessão. Como principal resultado, este trabalho tem a proposta de cinco contramedidas a este tipo de ataque. As contramedidas são baseados em modificações em algoritmos já conhecidos na literatura. Com as contramedidas propostas, foi analisado o tempo de execução de cada algoritmo e a variância do tempo de execução em relação ao grau do polinômio de entrada de cada algoritmo. Para todas as contramedidas os resultados foram superiores às propostas originais, no que diz respeito a variância

do tempo. Reduzindo consideravelmente o tempo de execução para cada estratégia de computar as raízes de um polinômio em uma extensão de corpo finito. Consequentemente reduzindo a variação do tempo de execução do algoritmo de decodificação do criptossistema de McEliece, e tornando a implementação segura contra ataques de efeito colateral baseado na variação do tempo.

Considerações Finais

Neste trabalho foi apresentado cinco contramedidas a um ataque de canal lateral baseados na variação de tempo para o criptossistema de McEliece. Todas essas contramedidas foram implementadas e medidas a sua eficacia contra este tipo de ataque. De modo que o objetivo deste trabalho foi alcançado. De modo que uma nova implementação do algoritmo de decodificação do McEliece pode ser realizada, que seja segura contra ataques de efeito colateral. Construindo protocolos criptograficamente seguros contra ataques realizados em computadores clássicos e também resistente a ataques realizados em computadores quânticos.

Por fim, trabalhos futuros são propostos, como implementar o novo método de euclides proposto recentemente que tem tempo de execução constante. Adicionalmente, realizar a medição das contramedidas em um sistema dedicado, tornando a medição mais precisa. Adicionalmente, realizar o estudo de diferentes cenários de ataques para o criptossistema de McEliece. **Palavras-**

chave: Ataque de efeito colateral. Criptografia pós-quantica. Criptografia baseada em códigos de correção de erro. Raizes de polinomios.

ABSTRACT

In the last few years, post-quantum cryptography has received much attention. National Institute Standards and Technology (NIST) is running a competition to select some post-quantum schemes as standard. As a consequence, implementations of post-quantum schemes have become important and with them side-channel attacks. In this work, we show a timing attack on a code-based scheme which was submitted to the NIST competition. This timing attack recovers secret information because of a timing variance in finding roots in a polynomial. We present five algorithms to find roots that are protected against timing exploitation.

Keywords: Side-channel Attack. Post-quantum Cryptography. Code-based Cryptography. Roots Finding.

LIST OF FIGURES

Figure 1 – Code-based cryptography main idea. 35
Figure 2 – Measurements time for the five methods presented. 57
Figure 3 – Measurements cycles for methods presented in Chapter 4 59
Figure 4 – Comparison of original implementation and our proposal. 60

LIST OF SYMBOLS

$\mathbb{N}$	Set of natural numbers
$\mathbb{F}_q$	Finite field of size q
$\mathbb{F}_{2^m}$	Binary finite field of size 2^m
$\mathbb{F}_q^n$	Vector space over finite field $\mathbb{F}_q$
$\{0,1\}^k$	Vector of 0's and 1's with size k
$w_h(x)$	Hamming weight of x

CONTENTS

1	**INTRODUCTION**	**23**
1.1	OBJECTIVES	24
1.1.1	**Specific objectives**	**24**
1.2	METHODOLOGY	25
1.3	SCIENTIFIC CONTRIBUTION	26
1.4	ORGANIZATION	26
2	**MATHEMATICAL BACKGROUND**	**27**
2.1	ALGEBRAIC STRUCTURES	27
2.2	CODING THEORY	27
2.2.1	**Linear codes**	**28**
2.2.2	**Goppa codes**	**28**
2.2.3	**Patterson decoding algorithm**	**29**
2.3	PUBLIC-KEY CRYPTOGRAPHY	29
2.3.1	**Key encapsulation mechanisms**	**30**
2.4	SIDE-CHANNEL ATTACKS	30
2.4.1	**Timing side-channel attacks**	**30**
2.4.2	**Timing side-channel attacks on code-based cryptosystems**	**31**
2.5	LITERATURE REVIEW	32
3	**CODE-BASED CRYPTOGRAPHY**	**35**
3.1	MCELIECE CRYPTOSYSTEM	35
3.1.1	**Definitions**	**36**
3.1.2	**Decryption**	**37**
3.2	BIGQUAKE	38
3.2.1	**Submission overview**	**38**
3.2.2	**Timing side-channel attack**	**39**
3.2.2.1	*Implementation remarks*	*41*
4	**ROOT FINDING TECHNIQUES AND COUNTERMEASURES**	**43**
4.1	EXHAUSTIVE SEARCH	43
4.2	BERLEKAMP TRACE ALGORITHM	45
4.3	LINEARIZED POLYNOMIALS	47
4.4	SUCCESSIVE RESULTANT ALGORITHM	49
4.5	RABIN ALGORITHM	51
5	**COMPARISON**	**53**
5.1	COMPLEXITY ANALYSIS	53

5.2 PERFORMANCE ANALYSIS 53
5.3 TIME VARIANCE ANALYSIS 57
5.4 SECURITY OVERVIEW 60

6 FINAL CONSIDERATIONS 63
6.1 FUTURE WORK 64

BIBLIOGRAPHY 65

A IMPLEMENTATION REMARKS ON BIGQUAKE ATTACK 69

1 INTRODUCTION

Communications through electronic devices require privacy. This privacy between two parts is made with key agreements and key encapsulation protocols, or public-key algorithms. For several years, these protocols were designed over classical asymmetric cryptography, which security is based on number theory problems. Nowadays, integer factorization and discrete logarithm, typical examples of such problems, are considered secure. However, the quantum algorithm proposed by Shor (SHOR, 1997) is able to solve these numerical problems with a polynomial-time algorithm in a quantum computer. Besides, the recent and fast advances in quantum computing make necessary the study of new cryptography primitives, which are resistant to Shor's algorithm.

In recent years, the area of post-quantum cryptography has received considerable attention, mainly because of the call by the National Institute of Standards and Technology (NIST) for the standardization of post-quantum cryptosystems. On this call, NIST did not give restrictions about specific hard problems. However, most of the submitted schemes for Key Encapsulation Mechanisms (KEM) are lattice- and code-based. The latter is centered around coding theory and includes one of the oldest unbroken cryptosystems, namely McEliece cryptosystem (MCELIECE, 1978).

This classical algorithm uses an error-correcting code that can recover errors added to a message. This process is achieved through redundancy added to the original message. Using the idea behind coding theory, and protecting some parts of the code, only parties with knowledge of the code structure are able to recover the original message. Hence, we can construct schemes based on coding theory, which are safe against quantum and classical attacks. However, the scheme implementation may not be secure. One of the requirements for those proposals is that they are resistant to all known cryptanalysis methods. In particular, cryptosystems need to avoid side-channel attacks.

There are many different type of side-channel attacks that can be applied to code-based cryptosystem. As an example, an attacker can measure the execution time of the operations performed by an algorithm and, based on these time variations, estimate some secret information of the scheme. Although the attack scenario is non-trivial, side-channel attacks are a dangerous mechanism that a cryptosystem designer should consider.

In code-based cryptography, timing attacks on the decryption process are mostly made during the retrieval of the Error Locator Polynomial (ELP). The attack is usually made in the process of evaluating the polynomials, performed to find its roots. This attack was demonstrated firstly in (SHOUFAN et al., 2009) and later in an improved version in (BUCERZAN et al., 2017).

In (STRENZKE, 2012), the authors present a survey of algorithms and compare their performances to find roots in code-based cryptosystems efficiently. However, the author only shows timings speedup in different types of implementations. Additionally, they select the one which has the least timing variability. In other words, the authors do not present an algorithm to

find the roots in constant time and therefore eliminate the attack, as remarked in (STRENZKE, 2013).

The algorithms presented in (STRENZKE, 2012) are not designed with constant behavior of the implementations in mind. The authors present two optimizations in the exhaustive search method, but these do not affect time variations while the algorithms are in execution. The author further proposes other improvements, some of them in the classical Berlekamp Trace Algorithm (BERLEKAMP, 1970) and also in the algorithm proposed in (FEDORENKO; TRIFONOV, 2002). However, none of these implementations focus on constant-time behaviors to protect the application and thus may leak sensitive information in the decoding process of the McEliece cryptosystem.

The root-finding implementations presented in (CHOU, 2017; BERNSTEIN; CHOU; SCHWABE, 2013) use Fast Fourier Transforms (FFT), and, while efficient, they are built and optimized for a specific finite field. We are interested in proposing a more generic implementation that does not require specific optimizations on the underlying finite field arithmetic. Additionally, their approach takes advantage of computer architecture and uses the fact that it can evaluate multiple points in parallel. We are also interested in approaches that could avoid side-channel attacks in any architecture.

In this work, to evidence the threat of a timing side-channel attack, we present an implementation of the Strenzke's attack over a NIST Round 1 submission, the BIGQUAKE. The main focus of this work is to propose countermeasures to make the execution time of the aforementioned algorithms constant. One of the countermeasures is to write the algorithms iteratively, eliminating all recursions. Additionally, we propose the use of probabilistic algorithms to achieve a secure root computation. We also use permutations and simulated operations to mask possible measurements of the side effects of the data being measured.

1.1 OBJECTIVES

The main goal of this work is to find new alternatives to perform the decoding process of McEliece Cryptosystem safely, avoiding timing side-channel attacks. To achieve this, we are interested in building a constant time to compute the roots of Error Locator Polynomial or remove the relation between the polynomial to be factorized and the execution time of the algorithm.

1.1.1 Specific objectives

i. Perform a timing side-channel attack against a code-based cryptosystem which has non-constant time root extraction;

ii. Select methods to compute the roots of a polynomial in code-based cryptosystems;

iii. Measure the time variation of methods selected in the previous item;

iv. Propose strategies to achieve a secure way to compute roots;

v. Evaluate new time variations of our proposals.

1.2 METHODOLOGY

In order to accomplish the aforementioned goals, we present the methodology used in this work.

1. Code-based cryptosystems overview: We first study the theoretical background needed to understand the code-based cryptosystems. Starting from the first proposal by McEliece until the NIST standardization submissions;

2. Literature review of timing side-channel attacks on code-based cryptosystems: The second step is to perform a literature review over side-channel attacks against code-based schemes. This literature review is constructed through a search over the main journals and conferences;

3. Perform a timing side-channel attack against a cryptosystem: to illustrate that a naive implementation is insecure against a timing side-channel attack, we perform an attack against a code-based scheme;

4. Literature review of root-finding methods: the main algorithms used to compute the roots of a polynomial in code-based cryptosystems are listed and individually studied;

5. Addition of different methods: Since the area of computing roots of a polynomial does not apply only to code-based cryptosystem, a search for methods which are not being used in code-based cryptography is done, and we list them together with the methods indexed in the previous item;

6. Propose countermeasures: Since the listed algorithms are not originally designed to prevent a side-channel attacker, we propose countermeasures in order to protect the implementation and avoid the leakage of sensitive information;

7. Implementation: in order to effectively measure the efficacy of our proposal, we implement the countermeasures presented to analyze the information leakage in our approaches.

8. Analysis: We use our implementation to measure the time variance of all studied methods, with and without the countermeasures proposed. Additionally, we present an analysis of the execution time of each algorithm with different parameters;

9. Summarize the obtained results: all results obtained are summarized, and a conclusion is drawn, defining the results obtained and the future works.

1.3 SCIENTIFIC CONTRIBUTION

The timing side-channel attack performed in Chapter 3 and the countermeasures proposed in Chapter 4 resulted in the following paper:

- MARTINS, D.; BANEGAS, G.; CUSTÓDIO, R. Don't Forget Your Roots: Constant-Time Root Finding over $\mathbb{F}_2^m$. In: International Conference on Cryptology and Information Security in Latin America. 2019. p. 109-129. Available in: https://doi.org/10.1007/978-3-030-30530-7_6.

This thesis includes direct text from this publication.

1.4 ORGANIZATION

The remainder of this thesis is organized as follows. In Chapter 2, we give a brief description of the mathematical background used throughout the text for a better comprehension of our work. The McEliece cryptosystem, BIGQUAKE submission, and the timing side-channel attack are presented in Chapter 3. In Chapter 4, the core of this thesis, we present five methods for finding roots over binary finite fields. We also include countermeasures for avoiding timing attacks. Chapter 5 provides a comparison of the number of cycles of the original implementation and the implementation with countermeasures, a performance analysis, and the security consideration of our proposals. At last, in Chapter 6, we conclude this thesis and discuss open problems.

2 MATHEMATICAL BACKGROUND

In this chapter, we present an overview of the needed concepts to better understand this thesis. First, we describe the main properties of polynomials over finite fields. After, we present an overview of coding theory. We explain the main structures and operations. Moreover, we present a fundamental work called Goppa Codes and review the state of the art of algorithms used to find roots in code-based cryptography.

2.1 ALGEBRAIC STRUCTURES

The goal of this work is to define a secure way to compute the roots of a polynomial over a binary finite field. In this section, we define it and present the main characteristics of it.

Definition 2.1 *Let $\mathbb{F}_{2^m}$ be a finite field with 2^m elements, and $f \in \mathbb{F}_{2^m}[x]$ be given by $f(x) = \sum_{i=0}^{t} g_i x^i$.*

Now we can define irreducible polynomials.

Definition 2.2 *Let $f(x) = \sum_{i=0}^{t} g_i x^i$ be a polynomial of degree t, f is an irreducible polynomial if f can not be factored into the product of two or more polynomial, with one of them being constant.*

Also, we define an monic polynomial.

Definition 2.3 *Let $f(x) = \sum_{i=0}^{t} g_i x^i$ be a polynomial of degree t, f is a monic polynomial if the leading coefficient equals to 1.*

Thus, a monic polynomial of degree t has the form $x^t + g_{t-1}x^{t-1} + \cdots + g_0$. Finally, we define the roots of a polynomial.

Definition 2.4 *Let $f(x) = \sum_{i=0}^{t} g_i X^i$ be a polynomial of degree t. The roots of f are all elements in $\mathbb{F}_{2^m}$ that the result of evaluation in f is 0.*

We can note that a root of a polynomial is equivalent to a factor of it. Since a polynomial it is composed by a multiplication of their factors. When we evaluate the polynomial with one factor, the corresponding factor on the product is equal to zero. Consequently, the results of the evaluations are equal to zero.

2.2 CODING THEORY

Coding theory is an engineering area that focuses on data transmission. Many communications channels are susceptible to the introduction of errors, for example, wireless connection. The error detection and correction techniques able us to recover the originally sent message. In this work, we study cryptography schemes that made use of an error correction

code to achieve key exchange between two parts. In the following sections, we give a brief introduction to the main concepts of error correction codes, and we also present the Goppa code, which is the linear code used to achieve key exchange in McEliece cryptosystem.

2.2.1 Linear codes

Definition 2.5 *Let $\mathbb{F}_q$ be a finite field with q elements. A linear $[n,k]$ code C is a subspace of dimension k of the vector space $\mathbb{F}_q^n$.*

The parameter k is the code dimension, and n represents the length. The code C is able to correct up to t errors. This recovery is given through the redundancy added in a message with size k, encoded in a codeword of size n. Messages in $\mathbb{F}_q^k$ are mapped to a unique codeword. One of the most used metrics in codes is the Hamming metrics, defined below.

Definition 2.6 *Let $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_n) \in C$ be two codewords. The Hamming distance between a and b is the number of positions in which they differ.*

Definition 2.7 *Let $a = (a_1, a_2, \ldots, a_n) \in C$ be a codeword. The Hamming weight of the codeword a is the number of non-zero positions.*

2.2.2 Goppa codes

Let $m,n,t \in \mathbb{N}$. A binary Goppa code $\Gamma(L,g)$ is defined by a polynomial $g(z) = \sum_{i=0}^{t} g_i z^i$ over $\mathbb{F}_{2^m}$ with degree t and supported by $L = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, such that $\alpha_i \in \mathbb{F}_{2^m}$ with $\alpha_i \neq \alpha_j$ for $i \neq j$, and $g(\alpha_i) \neq 0$ for all $\alpha_i \in L$ with g square-free. To a vector $c = (c_1, \ldots, c_n) \in \mathbb{F}_2^n$ we associate the syndrome polynomial

$$S_c(z) = \sum_{i=1}^{n} \frac{c_i}{z + \alpha_i}, \tag{2.1}$$

where $z + \alpha_i$ is invertible $\pmod{g(z)}$.

Definition 2.8 *The binary Goppa code $\Gamma(L,g)$ consists of all vectors $c \in \mathbb{F}_2^n$ such that*

$$S_c(z) \equiv 0 \pmod{g(z)}. \tag{2.2}$$

The parameters of a linear code are the size n, dimension k, and minimum distance d. We use the notation $[n,k,d]-$Goppa code for a binary Goppa code with parameters n,k and d. If the polynomial g which defines a Goppa code is irreducible over $\mathbb{F}_{2^m}$, the code is an irreducible Goppa code.

The length of a Goppa code is given by $n = |L|$ and its dimension is $k \geq n - mt$, where $t = \deg(g)$, and the minimum distance of $\Gamma(L,g)$ is $d \geq 2t+1$. The syndrome polynomial $S_c(z)$ can be written as:

$$S_c(z) \equiv \frac{w(z)}{\sigma(z)} \pmod{g(z)}, \tag{2.3}$$

where $\sigma(z) = \prod_{i=1}^{l}(z + \alpha_i)$ is the product over those $(z + \alpha_i)$ where there is an error in position i of c. This polynomial σ is the Error-Locator Polynomial (ELP).

A binary Goppa code can correct a codeword $c \in \mathbb{F}_2^n$, obscured by an error vector $e \in \mathbb{F}_2^n$ with Hamming weight $w_h(e)$ up to t, i.e., the numbers of non-zero entries in e is at most t. The way to correct errors is by using a decoding algorithm. For irreducible binary Goppa codes, we have three main alternatives for that. The extended Euclidean Algorithm (EEA) and the Berlekamp-Massey algorithm are out of the scope of interest for this work because they needed a parity-check matrix that has twice more rows than columns. The Patterson algorithm (PATTERSON, 1975), that are the focus of our work, can correct up to t errors with smaller structures, e.g., a smaller generator polynomial.

2.2.3 Patterson decoding algorithm

Patterson's algorithm is one of the most used decoding algorithms for Binary Goppa codes. The Algorithm 1 is the description of Patterson's algorithm. First, the algorithm starts by computing the inverse of the syndrome polynomial S_c. After that, we compute the square root of this inverse τ, and the odd and even parts of the error locator polynomial. Finally, using a and b we construct the error locator polynomial σ and we can reconstruct the error vector from the ELP.

Algorithm 1: Patterson decoding algorithm.

Data: S_c as syndrome polynomial and (L, g).
Result: Original message m of size k

1. Compute $\tau(z) = \sqrt{S_c^{-1}(z) + z}$;
2. Compute $b(z)$ and $a(z)$, so that $b(z)\tau(z) = a(z) \mod g(z)$, such that deg($a$)$\leq \lfloor \frac{t}{2} \rfloor$ and deg(b)$\leq \lfloor \frac{t-1}{2} \rfloor$;
3. Compute the error locator polynomial $\sigma(z) = a^2(z) + zb^2(z)$ and deg$(\sigma) \leq t$;
4. The position in L of the roots of σ defines the error vector e;
5. Compute the plaintext $m = c \oplus e$;
6. **return** m;

It is important to remark that the heaviest part of Patterson's algorithm is computing roots of σ. Patterson's algorithm can correct up to t errors. The complete proof of the algorithm correctness was presented in full detail in (PATTERSON, 1975). A suitable explanation of the algorithm workflow is present in (BARRETO; LINDNER; MISOCZKI, 2010).

2.3 PUBLIC-KEY CRYPTOGRAPHY

Public-key cryptography was first introduced by Diffie and Hellman (DIFFIE; HELLMAN, 1976) as a new concept to cryptography area. After that, Rivest, Shamir, and Adleman

propose one of the most important public-key cryptosystems that still is massively used on different cryptography applications (RIVEST; SHAMIR; ADLEMAN, 1978). The main idea of asymmetric cryptography was to associate a public key to a private key. The public key is distributed for and used for encryption; thus, given a ciphertext, only who knows the private key can recover the plain text. Using this idea, we present in the next section the idea of key encapsulation mechanisms.

2.3.1 Key encapsulation mechanisms

Most of the public-key algorithms are clumsy to be used in data transmission. Thus, a key encapsulation mechanism could be used to exchange a symmetric key, and the data transmission could be done using an efficient algorithm, like AES (DAEMEN; RIJMEN, 2013). Key encapsulation mechanisms are widely used in many protocols, for example, TLS (Transport Layer Security) protocol. It makes use of digital certificates to guarantee the authenticity of the server and provide a private connection between the browsers and servers.

A Key encapsulation mechanism is defined by a 3-tuple of algorithms `(KeyGen, Enc, Dec)`, where `KeyGen` is a probabilistic algorithm that receives a security parameter outputs a key pair. The `Enc` is an encapsulation algorithm that receives as input a public key and a plain text and returns a ciphertext. The encapsulation process could be a probabilistic algorithm. The `Dec` is a decapsulation algorithm, which takes as input the ciphertext and the private key and returns the plaintext or a failure. In the Chapter 3, we present an implementation of a Key encapsulation mechanism.

2.4 SIDE-CHANNEL ATTACKS

A side-channel attack is any attack where the attacker acquires sensitive information from the implementation of the cryptosystem. There are several types of side-channel attacks, and almost all of then require extensive knowledge over the target cryptosystem. Even the standard cryptography primitives, like the RSA (RIVEST; SHAMIR; ADLEMAN, 1978), are susceptible to side-channel attacks (KOCHER, 1996).

Similar to traditional cryptography, quantum resistance cryptography is susceptible to most side-channel attacks. In the case of the code-based cryptosystems, one of the most dangerous side-channel attacks is based on the time variance on the execution of the operations. This attack is called timing side-channel attack.

2.4.1 Timing side-channel attacks

A timing side-channel attack is an attack where it is possible to infer information from the time execution of a cryptographic algorithm. To better understand how a timing side-channel attack works, we present Example 2.1.

Example 2.1 *Let us suppose that an attacker has access to the execution time of a password validation (for instance, we use a four-digit password). This validation was made through the naive implementation presented in the Listings 2.1*

Listing 2.1 – Naive implementation of a password check

```
// Dumby password validation
int passCheck(char * input_pass) {
    int i = 0;
    // Iterates for all characters
    while (i < 4) {
        // Verify if are equal
        if (input_pass[i] == getPassAt(i)) {
            continue;
        } else {
            // If different, returns error code
            return -1;
        }
    }
    return 0;
}
```

An attacker with knowledge of the implementation and with access to the time wasted to verify a password, could perform a simple side-channel attack. First, setting a password attempt to "AAAA", after that, flip the first character to all possible letters, and assume that the correct was the one that spent more time. The attacker conjectures this because, if the first character is right, the password verification will check at least one more position. After repeating this for all remaining letters, the attacker recovers the secret password.

We present this toy example to give an idea of how an attacker, who knows the implementation details can infer information about the execution time of an algorithm. In more complex cryptosystems, this inference is not that simple. However, it is still possible to steal sensitive information measuring the decoding execution time.

2.4.2 Timing side-channel attacks on code-based cryptosystems

Equally to many other cryptosystems, code-based schemes are susceptible to side-channel attacks. Since the sensitive artifact in a code-based cryptosystem are the properties that able the decoding process, the decryption step is the target of side-channel attacks. Additionally, there are two different types of attacks, with focus on recover private key and to recover the original message. In this work, we focus on timing side-channel attacks that aim to recover the plain text.

One of the most popular idea on timing side-channels attacks over code-based schemes is based on the fact that a cipher message with more errors spends more timing decoding. In a scenario when an attacker is able to measure the time wasted on the decoding algorithm, and also are able to modify the message passed to the decoder, this attacker could measure the time variation and infers sensitive information, such as the original message.

The authors in (SHOUFAN et al., 2009) present the idea of inferring information on the time variance when we flip a bit over a ciphertext. This idea is based on the fact that if we flip a correct bit (i.e. a position where an error was added), the algorithm will have fewer errors to correct and will return early. The opposite occurs when we flip the wrong bit. Thus, the algorithm will return lately. The attack presented in Chapter 3 is based on this attack.

2.5 LITERATURE REVIEW

The decoding process of the McEliece cryptosystem, presented in the next chapter, is the most sensitive part of the algorithm. The usage of the private key demands that this algorithm does not leak any information about such key, or even, any information about the error added to the message. The authors in (BUCERZAN et al., 2017) show that a time deviation on the root-finding process in Patterson's decoding algorithm could open avenues for side-channel attacks.

In this section, we present the related works as a results of our literature review. Our main goal was to propose a method to achieve root computation of a polynomial over a binary field. The main focus of this root computation is to avoid timing side-channel attacks over code-based cryptosystems. Thus, we start our literature review by searching for root-finding methods that are currently used in code-based schemes.

A direct way to find all roots of a polynomial is an exhaustive search. In summary, it just tests all possible elements and checks if they are roots. The authors in (STRENZKE, 2012) show an optimization to this approach. They divide the original polynomial when a root is found. This reduces the size of the polynomial and speeds up the exhaustive search in about 30%. Although this algorithm is more efficient than the original one, it does not prevent side-channel attacks and still has exponential execution time.

Berlekamp, in (BERLEKAMP, 1970), proposed an alternative method to finding-roots. The classical Berlekamp Trace Algorithm computes the roots of a polynomial efficiently in polynomial time. The authors in (STRENZKE, 2012) present an optimized version by returning to the recursion before the original algorithm. Another approach to compute roots was proposed in (FEDORENKO; TRIFONOV, 2002), and was generalized in (SKACHEK; ROTH, 2008; BISWAS; HERBERT, 2009). However, all of these approaches do not present constant behaviour on execution time, and consequently are susceptible to a side-channel attack.

More recently, the authors in (PETIT, 2014) present a different algorithm to compute the roots of a polynomial. Lately, generalized in (DAVENPORT; PETIT; PRING, 2016), the Successive Resultants Algorithm (SRA) is a polynomial method, with a different approach to

finding roots in a polynomial over any finite field. Nevertheless, no analysis of execution time deviation was made.

All approaches presented in this section were root-finding techniques currently in use in code-based cryptosystems. Nonetheless, it was well known that Cantor and Zassenhaus proposed one of the most efficient methods for computing roots over a polynomial in 1981 (CANTOR; ZASSENHAUS, 1981). Notwithstanding its efficiency, to the best of our knowledge, it was not used in any code-based cryptosystems. For this work, we consider the Rabin algorithm, which is adapted to fields with even characteristic.

3 CODE-BASED CRYPTOGRAPHY

The rising of quantum computing in the last few years turned the attention of researchers to schemes that use cryptography primitives different from discrete logarithm and integer factorization. As mentioned in Chapter 1, code-based cryptosystems are those which use fundamental aspects of coding theory to add redundancy into a plain text, to after that, intentionally add some errors and recover the plain text.

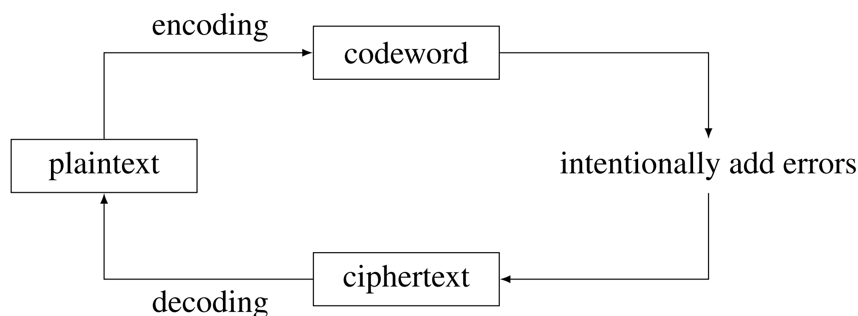
In this chapter, we explain the first cryptosystem based on coding theory, the McEliece Cryptosystem. Furthermore, we present the code-based scheme BIGQUAKE, that was submitted to the first round of the NIST standardization process. Then, we show the timing side-channel attack performed in the reference implementation of this scheme.

3.1 MCELIECE CRYPTOSYSTEM

Robert J. McEliece proposed in 1978 the first cryptosystem based on coding theory. The McEliece cryptosystem is based on Goppa codes and has three main algorithms, i.e.: key Generation, encryption, and decryption (MCELIECE, 1978). Figure 1 shows the main idea of the McEliece Cryptosystem and most of the schemes based on coding theory. Given a plaintext and a public encoding function, anyone can generate a codeword and intentionally add some errors to obtain a ciphertext. From this ciphertext, only parties with the knowledge of the code structure can perform a decoding procedure and recover the plaintext.

The security of the McEliece cryptosystem has remained stable based on two well know assumptions. The first assumption is the hardness of generic decoding, that is NP-complete and is also considered hard on average (BERLEKAMP; MCELIECE; TILBORG, 1978). The second assumption with regards to the security of the McEliece scheme is the indistinguishability of the code. It is not old as the first one, but it relates to old problems of algebraic coding theory. According to (FAUGERE et al., 2013), it is also valid to consider this assumption.

The original parameters were designed for 64 security bits, but it easily scales up to



Source – the author.

Figure 1 – Code-based cryptography main idea.

provide an ample security margin against attackers (CANTEAUT; SENDRIER, 1998). Furthermore, the McEliece cryptosystem has been subjected to many structural attacks, and up to today the original proposal is considered secure (BERNSTEIN; LANGE; PETERS, 2008). It is important to note that many of these attacks are based on the same strategy; the information set-decoding. This kind of attack does not affect schemes based on Goppa codes. However, many other families of codes suffer several drawbacks with this attack.

As proposed in (BERNSTEIN et al., 2017; BARDET et al., 2017) on the NIST standardization process, the McEliece cryptosystem, using binary Goppa codes, could be applied to conceive a KEM, as defined in Chapter 2. Based on this fact, we can generalize the cryptosystem in three algorithms: the key generation and encryption process are defined in Section 3.1.1 and Decryption in Section 3.1.2.

3.1.1 Definitions

In this section, we describe the three main algorithms of the McEliece cryptosystem based on irreducible binary Goppa codes. These algorithms are related to the generation of a Goppa code, the encoding process, and the decoding process, as presented in Chapter 2.

Algorithm 2 shows the key generation of McEliece. First, it starts by generating a binary irreducible Goppa polynomial g of degree t. This random selection is made by generating a random polynomial and testing if it is an irreducible polynomial. Second, it generates the support L as an ordered subset of $\mathbb{F}_{2^m}$ satisfying the root condition. Third, it computes the systematic form of H' using the Gauss-Jordan elimination algorithm. Steps four, five, and six compute the generator matrix from the previous systematic matrix and return secret and public key.

Algorithm 2: McEliece key generation.

Data: t, k, n, m as integers.
Result: pk as public key, sk as secret key.
1 Select a random binary Goppa polynomial g of degree t over $\mathbb{F}_{2^m}$;
2 Randomly choose n distinct elements of $\mathbb{F}_{2^m}$ that are not roots of g as the support L;
3 Compute the $k \times n$ parity check matrix H' according to L and g;
4 Bring H' to systematic form: $H_{sys} = [H'|I_{n-k}]$;
5 Compute generator matrix G from H_{sys};
6 **return** $sk = (L, g) \quad pk = (G)$;

Given a public key, generated by Algorithm 2 and a message m, Algorithm 3 shows the encryption process of McEliece. The process is efficient and straightforward, requiring only a random error vector e with $w_h(e) \leq t$ and multiplication of a vector by a matrix. After the multiplication, we intentionally add the error vector to the codeword obtained and return the ciphertext.

Algorithm 3: McEliece encryption.

Data: Public key $pk = G$, message $m \in \mathbb{F}_2^k$.
Result: c as ciphertext of length n.
1 Randomly choose an error vector e of length n with $w_h(e) \leq t$;
2 Compute $c = (m \cdot G) \oplus e$;
3 **return** c;

3.1.2 Decryption

Algorithm 4 describes the decryption part of McEliece. This algorithm consists of the removal of the applied errors using a decoding algorithm. First, we compute the syndrome polynomial S_c. Second, we recover the error vector e from the syndrome polynomial. Finally, we recover the plaintext m computing $c \oplus e$, i.e., the exclusive-or of the ciphertext and the error vector. We observe that in modern KEM versions of McEliece, $m \in \mathbb{F}_2^n$ is a random bit string used to compute a session key using a hash function. Hence, in moderns KEM, in the first k positions of m we have random data. Then, the amount of error would be moderated.

Algorithm 4: McEliece decryption.

Data: c as ciphertext of length n, secret key $sk = (L, g)$.
Result: Message m of size k
1 Compute the syndrome $S_c(z) \equiv \sum \frac{c_i}{z+\alpha_i} \pmod{g(z)}$;
2 Compute $\tau(z) = \sqrt{S_c^{-1}(z) + z}$;
3 Compute $b(z)$ and $a(z)$, so that $b(z)\tau(z) \equiv a(z) \pmod{g(z)}$, such that deg($a$)$\leq \lfloor \frac{t}{2} \rfloor$ and deg(b)$\leq \lfloor \frac{t-1}{2} \rfloor$;
4 Compute the error locator polynomial $\sigma(z) = a^2(z) + zb^2(z)$ and deg(σ) $\leq t$;
5 The position in L of the roots of $\sigma(z)$ define the error vector e;
6 Compute the plaintext $m = c \oplus e$;
7 **return** m;

In the decryption algorithm, Steps 2-5 are the description of Patterson's decoding algorithm (PATTERSON, 1975). This same strategy can be used in schemes based on the Niederreiter cryptosystem (NIEDERREITER, 1986). These schemes differ in their public-key structure, encryption, and decryption step, but both of them, in the decryption steps, decode the message from the syndrome. Another decoding algorithm can be applied to decode a message with errors. Still, the main focus of this work is on Patterson's algorithm, since it is the most used on McEliece cryptosystem with Goppa codes.

The roots of the ELP can be acquired with different methods. Although these strategies can be implemented with different forms, the implementations must not leak any timing information about their execution. This leakage can lead to a side-channel attack using time differences in the decryption algorithm, as we explore in a scheme in Section 3.2.

3.2 BIGQUAKE

In this section, we describe the BIGQUAKE submission, the usage of the McEliece in their protocol, and after that, we present a timing side-channel attack against the decapsulation process. The attack was originally proposed in (SHOUFAN et al., 2009). We use the fact that BIGQUAKE claims to be indistinguishability under adaptive chosen ciphertext attack (IND-CCA) to create our attack scenario and perform the attack proposal (BERNSTEIN; PERSICHETTI, 2018). Our attack was implemented based on BIGQUAKE implementation available in https://bigquake.inria.fr/files/2018/03/BIGQUAKE-source.tar.gz.

3.2.1 Submission overview

BIGQUAKE (BARDET et al., 2017) uses binary Quasi-cyclic (QC) Goppa codes in order to accomplish a KEM between two distinct parts. Instead of using binary Goppa codes, BIGQUAKE uses QC Goppa codes, which have the same properties as Goppa codes but allow smaller keys.

Let us suppose that Alice and Bob (A and B respectively) want to share a secret session key K using BIGQUAKE. Then Bob needs to publish his public key, and Alice needs to follow the encapsulation mechanism. After receiving the vector c from Alice, Bob executes the decapsulation process. If Bob succeeds, both parties obtain the same session secret key K. The security of the protocol is based on steps that involve the McEliece encryption performed by Alice and the McEliece decryption performed by Bob. The BIGQUAKE description is presented in Protocol 1.

The function $\mathcal{F}$ maps an arbitrary binary string as input and returns a word of weight t, i.e $\mathcal{F} : \{0,1\}^* \rightarrow \{x \in \mathbb{F}_2^n | w_h(x) = t\}$. The detailed construction of the function $\mathcal{F}$ can be found in subsection 3.4.4 in (BARDET et al., 2017). The cryptographic hash function $\mathcal{H} : \{0,1\}^k \rightarrow \{0,1\}^s$ maps an arbitrary binary string of size k to a fixed string of size s. The function $\mathcal{H}$ in the original implementation is SHA-3.

When Alice computes c_2, she is encrypting a message m, with error t and Bob's public key G. Additionally, Alice needs to compute exclusive-or between the hash of the error and the message. Also, she needs to compute the hash of the message. These last two values are used on Bob's side to verify if both sides are computing the same session key.

On Bob's side, after he receives c, he starts the decoding process on c_2, and only he is able to perform this correctly for the reason that Alice uses Bob's public key on the encapsulation process. To complete the key encapsulation mechanism, Bob uses the decoded e' to check against c_1 and c_2 if both sides are agreeing in the same session key.

The security of BIGQUAKE relies on the same security as other McEliece cryptosystems that uses Goppa Codes: the syndrome decoding problem and the indistinguishability of Goppa codes (BARDET et al., 2017). These two problems are well studied in the literature, and it is safe to use them as the security assumption of a cryptosystem (BERNSTEIN; LANGE; PE-

Protocol 1: BIGQUAKE Key Encapsulation Mechanism between Alice and Bob

Inputs. Bob's public key as a generator matrix G

Goal. Parties jointly agree the same session key K

The protocol:

1. **Encapsulation.**
   (a) Alice generates a random $m \in \mathbb{F}_2^s$;
   (b) Generate $e \leftarrow \mathcal{F}(m)$;
   (c) Alice sends $c \leftarrow (m \oplus \mathcal{H}(e), H \cdot e^T, \mathcal{H}(m))$ to Bob;
   (d) The session key is defined as: $K \leftarrow \mathcal{H}(m, c)$.
2. **Decapsulation.**
   (a) Bob receives $c = (c_1, c_2, c_3)$;
   (b) Using the secret key, Bob decodes c_2 to e' with $w_h(e') \leq t$ such that $c_2 = H \cdot e'^T$;
   (c) Bob computes $m' \leftarrow c_1 \oplus \mathcal{H}(e')$;
   (d) Bob computes $e'' \leftarrow \mathcal{F}(m')$;
   (e) If $e'' \neq e'$ or $\mathcal{H}(m') \neq c_3$ then Bob aborts;
   (f) Else, Bob computes the session key: $K \leftarrow \mathcal{H}(m', c)$.

TERS, 2008; FAUGERE et al., 2013). Additionally, other schemes on the NIST standardization rely on their security on these same properties(BERNSTEIN et al., 2017).

As mentioned in Chapter 2, a secure scheme could suffer attacks on their implementation. The BIGQUAKE has well know semantic security assumptions. Nevertheless, its implementation has several problems against a timing side-channel attack. In the next section, we present our attack against the reference implementation of the BIGQUAKE.

3.2.2 Timing side-channel attack

In (SHOUFAN et al., 2009), the attack exploits the fact that flipping a bit of the error e changes the Hamming weight w and per consequence the timing for its decryption. If we flip a position that contains an error ($e_i = 1$) then the error will be removed and the time of computation will be shorter. However, if we flip a bit in a wrong position ($e_i = 0$) then it will add another error, and it will increase the decryption time. The attack described in (BUCERZAN et al., 2017) exploits the root-finding in the polynomial ELP. It takes advantage of sending ciphertexts with fewer errors than expected, which generates an ELP with a degree less than t, resulting in less time for finding roots. We explore both ideas applied to the implementation of BIGQUAKE.

Algorithm 5 is the direct implementation of the attack proposed in (SHOUFAN et al.,

2009). We use this attack on BIGQUAKE to show that implementations with the usual root-finding process are vulnerable to remote timing attacks.

Algorithm 5: Attack on root extraction of error locator polynomial of the BIGQUAKE.

```
   Data: n-bit ciphertext c, t as the number of errors and precision parameter M
   Result: Attempt to obtain an error vector e hidden in c.
 1 e ← [0,...,0];
   /* iterate over all bits on the ciphertext                          */
 2 for i ← 0 to n − 1 do
 3 |   A ← 0;
 4 |   c' ← c ⊕ setBit(n, i);
 5 |   time_m ← 0;
   |   /* perform M decryption operations measuring the average
   |      execution time                                                */
 6 |   for j ← 1 to M do
 7 |   |   time_s ← time();
 8 |   |   decrypt(c');
 9 |   |   time_e ← time();
10 |   |   time_m ← time_m + (time_e − time_s);
11 |   end
12 |   A ← time_m/M;
13 |   T ← (A, i);
14 end
   /* select the errors position according whose small average
      execution time                                                    */
15 Sort T in descending order of A;
16 for k ← 0 to t − 1 do
17 |   index ← T[k].i;
18 |   e[index] ← 1;
19 end
20 return e;
```

After finding the position of the errors, one needs to verify if the error e' found is the correct one, and then recover the message m. In order to verify for correctness, one can check e' by computing $\mathcal{H}(e) \oplus \mathcal{H}(e') \oplus m = m'$ and if c_3 is equal to $\mathcal{H}(m')$. As mentioned early in this section, the ciphertext is composed by $c = (m \oplus \mathcal{H}(e), H \cdot e^T, \mathcal{H}(m))$ or $c = (c_1, c_2, c_3)$.

In our attack, we select the precision parameter M as 500 since it shows the more precise results while maintains a relative level of efficacy. In our tests, it takes ≈ 17 minutes for recovering one correct error vector and, consequently, compute the session key. The attack was performed on an Intel® Core(TM) i7-4500U CPU @ 1.80GHz. The source code of this attack are available on https://git.dags-project.org/gustavo/roots_finding/src/master/attack_bigquake.

3.2.2.1 Implementation remarks

Some small corrections on the BIGQUAKE reference implementation were made to perform the attack correctly. First, the correct initialization of the error vector e with zeros in all positions. The original implementation does not fill all error positions with zero because the `malloc` function does not fill the positions with zero (C++ Standards Committee, 2014). We also modify the decryption failure condition. The original algorithm failure when an additional error are inserted. We removed the part that causes this failure on the attack implementation.

Additionally, we notice that the original implementation of BIGQUAKE uses log and antilog tables for computing multiplications and inversions. These look-up tables give a speedup in those operations. However, this approach is subject to cache attacks in a variation of (BRUINDERINK et al., 2016), where the attacker tries to induce cache misses and infer the data.

Since we want to avoid the use of look-up tables, we made a constant-time implementation for multiplication and inversion, using a similar approach as (CHOU, 2017). In order to illustrate that, Listing A.3 shows the multiplication in constant-time between two elements over $\mathbb{F}_{2^{12}}$ followed by the reduction of the result by the irreducible polynomial $f(x) = x^{12} + x^6 + x^4 + x + 1$. Further, the inversion in finite fields can be computed by raising an element a to the power $2^m - 2$, i.e., a^{2^m-2}, as presented on Listing A.4. These operations and the others implementation remarks are present on Appendix A.

The performed attack show us a naive implementation of root-finding enables a side-channel attacker to find the secret plain message on BIGQUAKE submission. In the next chapter, we propose five alternatives to construct a safe root-finding algorithm and consequently a secure McEliece implementation.

4 ROOT FINDING TECHNIQUES AND COUNTERMEASURES

As argued, the leading cause of information leakage in the decoding algorithm is the process of finding the roots of the ELP. In general, the time needed to find these roots varies, often depending on the roots themselves. Thus, an attacker who has access to the decoding time can infer these roots, and hence get the vector of errors e. One example of this attack is presented in Section 3.2, where we perform a side-channel attack over the decryption part of a cryptosystem, and we recover the secret exchange between two parts of the protocol in an acceptable time on a conventional laptop.

Factoring polynomials is a well-studied problem in the finite fields area. More recently, with the rising of the code-based area, some works deal with the factorization problem on the decoding process in order to avoid timing side-channel attacks (SHOUFAN et al., 2009; BUCERZAN et al., 2017). In our literature review, we select five methods used on code-based cryptosystems to analyze and propose modifications in their implementation to make them available on cryptosystems applications.

The adjustments were made in the following algorithms to find roots: exhaustive search, linearized polynomials, Berlekamp Trace Algorithm (BTA), and Successive Resultant Algorithm (SRA). Also, we present the Cantor-Zassenhaus method, as it has a probabilistic algorithm that makes use of a random selection of polynomials in its execution. We make a security analysis over this algorithm to evaluate their time variance on a side-channel attack scenario. The description of each algorithm is given in the next sections, and the analysis is present in Chapter 5.

As the main focus of this works relies on the root computation of a polynomial over a finite field in cryptographic applications, we restrict our scope to polynomials over binary fields. Also, we assume that we want to compute the roots of a squarefree polynomial. Because of the nature of the ELP, it is composed of the code locators, and each error represents one factor. However, if this was not the nature of the polynomial, we can achieve a squarefree easily, see (VON ZUR GATHEN; PANARIO, 2001).

4.1 EXHAUSTIVE SEARCH

The exhaustive search, also called Chien Search (CHIEN, 1964), is a direct method, in which the evaluation of f for all the elements in $\mathbb{F}_{2^m}$ is performed. A root is found when the evaluation result is zero. This method is acceptable for small fields and can be made efficient with a parallel implementation. The greatest drawback of this method is its asymptotic complexity, and this is the reason it is not widely used. However, we consider it still relevant, since some NIST proposals are based on a small finite field, enabling their usage.

Algorithm 6 describes the exhaustive search. For all elements in $\mathbb{F}_{2^m}$, we evaluate the polynomial and check if it is a root or not, and this method leaks information when a root is found. This leakage occurs because whenever we found, i.e., $dummy = 0$, an extra operation is

performed by adding the root found on the returning list R. In this way, the attacker can infer this from this additional time that a root was found, providing ways to obtain data that should be secret.

Algorithm 6: Exhaustive search algorithm for finding roots of a univariate polynomial over $\mathbb{F}_{2^m}$.

Data: p a univariate polynomial over $\mathbb{F}_{2^m}$ with t roots, $A = [a_0, \ldots, a_{n-1}]$ all elements in $\mathbb{F}_{2^m}$, $n = 2^m$ the length of A.
Result: R a set of roots of p.

```
1 R ← ∅;
2 for i ← 0 to n−1 do
3 |   dummy ← p(A[i]);
4 |   if dummy == 0 then
5 |   |   R.add(A[i]);
6 |   end
7 end
8 return R;
```

One solution to avoid this leakage is to permute the elements of vector A before the evaluation. Using this technique, an attacker can identify the extra operation, but without learning any secret information. In our case, we use the Fisher-Yates shuffle (BLACK, 1998) for shuffling the elements of vector A. In (WANG; SZEFER; NIEDERHAGEN, 2018), the authors show an implementation of the shuffling algorithm, which is safe against timing attacks. Thus, we can build and exhaustive search without leaking information against a side-channel attack. Line 1 on Algorithm 7 shows the permutation call of the elements and the computation of the roots.

Algorithm 7: Exhaustive search algorithm with a countermeasure for finding roots of an univariate polynomial over $\mathbb{F}_{2^m}$.

Data: p a univariate polynomial over $\mathbb{F}_{2^m}$ with t roots, $A = [a_0, \ldots, a_{n-1}]$ all elements in $\mathbb{F}_{2^m}$, $n = 2^m$ the length of A.
Result: R a set of roots of p.

```
1 permute(A);
2 R ← ∅;
3 for i ← 0 to n−1 do
4 |   dummy ← p(A[i]);
5 |   if dummy == 0 then
6 |   |   R.add(A[i]);
7 |   end
8 end
9 return R;
```

Using this approach, we add one extra step to the algorithm. However, this permutation blurs the sensitive information of the algorithm, making the usage of Algorithm 7 slightly harder for the attacker to acquire timing leakage. In this case, we want to avoid the addition of an `else`

clause adding the elements that are not a root in another list. This addition will reduce the time variance on the execution of the algorithm, but it could facilitate cache attacks (BERNSTEIN, 2005).

4.2 BERLEKAMP TRACE ALGORITHM

Our second strategy applies to the Berlekamp factoring algorithm (BERLEKAMP, 1970). Berlekamp presents an efficient algorithm to factor a polynomial, which can be used to find its roots. We call this algorithm BTA - *Berlekamp Trace Algorithm* since it relies on the trace function properties to split a polynomial into two small polynomials. This classical algorithm is one of the most used on code-based cryptosystems for the reason that it has lower complexity when compared with exhaustive search (BISWAS; HERBERT, 2009).

The trace function is defined as $Tr(x) = x + x^2 + x^{2^2} + \cdots + x^{2^{m-1}}$. It is possible to change BTA for finding roots of a polynomial p using $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ as a standard basis of $\mathbb{F}_{2^m}$, and then computing the greatest common divisor between $p(x)$ and $Tr(\beta_0 \cdot x)$. After that, BTA performs two recursive calls; one with the result of gcd algorithm and the other with the remainder of the division $p(x)/\gcd(p(x), Tr(\beta_i \cdot x))$ for successive i's. On the next call, the BTA must use a different basis element from the previous one.

The base case is when the degree of the input polynomial is smaller than or equal to one. In this case, BTA returns the root by getting the independent term of the polynomial. In summary, it is a divide and conquer like algorithm since it splits the task of computing the roots of a polynomial p into the roots of two smaller polynomials. Some improvements can be made on BTA with a hybrid version, i.e., when the degree of the polynomial is two, we can use a different algorithm to find its factors, as presented in (STRENZKE, 2012). As efficiency was not the focus of this work, we do not consider this approach. All steps of the Berlekamp trace algorithm are described in Algorithm 8.

Algorithm 8: Berlekamp Trace Algorithm – $BTA(p, i) - rf$.

Data: p a univariate polynomial over $\mathbb{F}_{2^m}$ and i to select the basis element.
Result: The set of roots of p.
1 **if** $deg(p) \leq 1$ **then**
2 | **return** root of p;
3 **end**
4 $p_0(x) \leftarrow gcd(p(x), Tr(\beta_i \cdot x))$;
5 $p_1(x) \leftarrow p(x)/p_0(x)$;
6 **return** $BTA(p_0(x), i+1) \cup BTA(p_1(x), i+1)$;

As we can see, a direct implementation of Algorithm 8 has no constant execution time. The recursive behavior may leak information about the characteristics of roots in a side-channel attack. Additionally, in our experiments, we note that the behavior of the gcd with the trace function might result in a polynomial with the same degree. Therefore, BTA will divide this

input polynomial in a future call upon a different basis element. Consequently, there is no guarantee of a constant number of executions because we cannot control if a polynomial will be split or not.

In order to avoid the nonconstant number of executions and avoid timing side-channel attacks, we propose an iterative implementation of Algorithm 8, hereafter referred to $BTA - it$. In this way, our proposal iterates in a fixed number of iterations instead of calling itself until the base case. The main idea is not changed; we still divide the task of computing the roots of a polynomial p into two smaller instances. However, we change the approach of the division of the polynomial. Since we want to compute the same number of operations independent of the degree of the polynomial, we perform the gcd with a trace function for all basis in β, and choose a division that results in two new polynomials with a medium degree.

This new approach allows us to define a fixed number of iterations for our version of BTA. Since we always divide into two small instances, we need $t-1$ iterations to split a polynomial of degree t into t polynomials of degree 1. Then we just need to add a stack to control the polynomials to be divided. Algorithm 9 presents our new approach of the iterative $BTA - it$.

Algorithm 9: Iterative Berlekamp Trace Algorithm – $BTA(p) - it$.

Data: p a univariate polynomial over $\mathbb{F}_{2^m}$, t the number of expected roots.
Result: The set of roots of p.

1 $g \leftarrow \{p(x)\}$; `// The set of polynomials to be computed`
2 **for** $k \leftarrow 0$ **to** t **do**
3 $current = g.pop()$;
4 Compute $candidates = gcd(current, Tr(\beta_i \cdot x))\ for\ all\ \beta_i \in \beta$;
5 Select $p_0 \in candidates$ such as $p_0.degree \simeq \frac{current}{2}$;
6 $p_1(x) \leftarrow current/p_0(x)$;
7 **if** $p_0.degree == 1$ **then**
8 $R.add$(root of p_0)
9 **end**
10 **else**
11 $g.add(p_0)$;
12 **end**
13 **if** $p_1.degree == 1$ **then**
14 $R.add$(root of p_1)
15 **end**
16 **else**
17 $g.add(p_1)$;
18 **end**
19 **end**
20 **return** R

Algorithm 9 extracts a root of the polynomial when the variable *current* has a polynomial of degree one. If this degree is higher than one, then the algorithm needs to continue dividing the polynomial until it finds a root. The algorithm does that by adding the polynomial

in a stack and reusing this polynomial in a future division. The iterative BTA has a higher complexity when compared to the original approach. Nevertheless, with our new approach, we achieve a more constant time execution on the root-finding task, consequently reducing the knowledge obtained by an attacker. In Chapter 5, we analyze and compare our iterative BTA with other approaches.

4.3 LINEARIZED POLYNOMIALS

The third countermeasure proposed is based on linearized polynomials. The authors in (FEDORENKO; TRIFONOV, 2002) propose a method to compute the roots of a polynomial over $\mathbb{F}_{2^m}$, using a particular class of polynomials, called linearized polynomials. In (STRENZKE, 2012), this approach is a recursive algorithm which the author calls "`dcmp-rf`". In our solution, however, we present an iterative algorithm. First, we define linearized polynomials.

Definition 4.1 *A polynomial ℓ over $\mathbb{F}_{2^m}$ is a linearized polynomial if*

$$\ell(y) = \sum_i c_i y^{2^i}, \tag{4.1}$$

where $c_i \in \mathbb{F}_{2^m}$.

In addition, from (TRUONG; JENG; REED, 2001), we have Lemma 4.1 that describes the main property of linearized polynomials for finding roots.

Lemma 4.1 *Let $y \in \mathbb{F}_{2^m}$ and let $\alpha^0, \alpha^1, \ldots, \alpha^{m-1}$ be a standard basis of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$. If*

$$y = \sum_{k=0}^{m-1} y_k \alpha^k, \quad y_k \in \mathbb{F}_2 \tag{4.2}$$

and $\ell(y) = \sum_j c_j y^{2^j}$, then

$$\ell(y) = \sum_{k=0}^{m-1} y_k \ell(\alpha^k). \tag{4.3}$$

We call $A(y)$ over $\mathbb{F}_{2^m}$ an affine polynomial if $A(y) = \ell(y) + \beta$ for $\beta \in \mathbb{F}_{2^m}$, where $\ell(y)$ is a linearized polynomial. Using this definition, we can construct a method for finding the ELP roots. First, we present a toy example from (TRUONG; JENG; REED, 2001) to understand the idea behind finding roots using linearized polynomials.

Example 4.1 *Let us consider the polynomial $f(y) = y^2 + (\alpha^2+1)y + (\alpha^2+\alpha+1)y^0$ over $\mathbb{F}_{2^3}$ and α a primitive element in $\mathbb{F}_2[x]/(x^3+x^2+1)$. Since we are trying to find roots, we can write $f(y)$ equals to zero*

$$y^2 + (\alpha^2+1)y + (\alpha^2+\alpha+1)y^0 = 0$$

and after that, we can rewrite this expression as

$$y^2 + (\alpha^2+1)y = (\alpha^2+\alpha+1)y^0. \tag{4.4}$$

We can point that on the left hand side of Equation (4.4), $\ell(y) = y^2 + (\alpha^2+1)y$ is a linearized polynomial over $\mathbb{F}_{2^3}$ and Equation (4.4) can be expressed as

$$\ell(y) = \alpha^2 + \alpha + 1. \tag{4.5}$$

If $y = y_2\alpha^2 + y_1\alpha + y_0 \in \mathbb{F}_{2^3}$ then, according to Lemma 4.1, Equation (4.5) becomes

$$y_2\ell(\alpha^2) + y_1\ell(\alpha) + y_0\ell(\alpha^0) = \alpha^2 + \alpha + 1. \tag{4.6}$$

We can compute $\ell(\alpha^0), \ell(\alpha)$ and $\ell(\alpha^2)$ using the left hand side of Equation (4.4) and we have the following values

$$\begin{aligned} \ell(\alpha^0) &= (\alpha^0)^2 + (\alpha^2+1)(\alpha^0) = \alpha^2 + 1 + 1 = \alpha^2, \\ \ell(\alpha) &= (\alpha)^2 + (\alpha^2+1)(\alpha) = \alpha^2 + \alpha^2 + \alpha + 1 = \alpha + 1, \\ \ell(\alpha^2) &= (\alpha^2)^2 + (\alpha^2+1)(\alpha^2) = \alpha^4 + \alpha^4 + \alpha^2 = \alpha^2. \end{aligned} \tag{4.7}$$

A substitution of Equation (4.7) into Equation (4.6) gives us

$$(y_2 + y_0)\alpha^2 + (y_1)\alpha + (y_1)\alpha^0 = \alpha^2 + \alpha + 1. \tag{4.8}$$

Equation (4.8) can be expressed as a matrix in the form

$$\begin{bmatrix} y_2 & y_1 & y_0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}. \tag{4.9}$$

If one solves the linear system in Equation (4.9) then the results are the roots of the original polynomial given in Equation (4.4). From Equation (4.8), one observes that the solutions are $y = 110$ and $y = 011$. Furthermore, we can translate 110 and 011 to $\alpha+1$ and $\alpha^2+\alpha$. After all these steps, it is easy to check if we found the factors of $f(y)$. We just need to multiply $y + (\alpha+1)$ by $y + (\alpha^2+\alpha)$, thus we get

$$(y + \alpha + 1)(y + \alpha^2 + \alpha) = y^2 + (\alpha^2+1)y + (\alpha^2 + \alpha + 1) = f(y).$$

Fortunately, the authors in (FEDORENKO; TRIFONOV, 2002) provide a generic decomposition for finding affine polynomials. Thus we can transform the error locator polynomial in an affine polynomial. In their work, each polynomial in the form $F(y) = \sum_{j=0}^{t} f_j y^j$ for $f_j \in \mathbb{F}_{2^m}$ can be represented as

$$F(y) = f_3 y^3 + \sum_{i=0}^{\lceil (t-4)/5 \rceil} y^{5i}\left(f_{5i} + \sum_{j=0}^{3} f_{5i+2^j} y^{2^j}\right). \tag{4.10}$$

After that, we can summarize all the steps on Algorithm 10. The function on Step 21, "`generate(m)`", refers to the generation of the elements in $\mathbb{F}_{2^m}$ using Gray codes. See (SAVAGE, 1997) for more details about Gray codes. The linearized method differs from the

exhaustive search on the evaluation process; it is much more efficient to compute this evaluation because the polynomial is in linearized form. Algorithm 10 presents our countermeasure in the last steps of the algorithm, i.e., we added a dummy operation for blinding if $X[j]$ is a root of polynomial $F(x)$. The analysis of the linearized method and its countermeasure are presented in Chapter 5.

4.4 SUCCESSIVE RESULTANT ALGORITHM

In (PETIT, 2014), the authors present an alternative method for finding roots in $\mathbb{F}_{p^m}$. Later on, the authors explain better the method in (DAVENPORT; PETIT; PRING, 2016). The Successive Resultant Algorithm (SRA) relies on the fact that it is possible to find roots exploiting properties of an ordered set of rational mappings.

Given a polynomial f of degree d and a sequence of rational maps $K_1, \dots, K_t$, the algorithm computes finite sequences of length $j \leq t+1$ obtained by successively transforming the roots of f by applying the rational maps. The algorithm is as follows: Let $\{v_1, \dots, v_m\}$ be an arbitrary basis of $\mathbb{F}_{p^m}$ over $\mathbb{F}_p$. Then it is possible to define $m+1$ functions $\ell_0, \ell_1, \dots, \ell_m$ from $\mathbb{F}_{p^m}$ to $\mathbb{F}_{p^m}$ such that

$$\begin{cases} \ell_0(z) = z \\ \ell_1(z) = \prod_{i \in \mathbb{F}_p} \ell_0(z - iv_1) \\ \ell_2(z) = \prod_{i \in \mathbb{F}_p} \ell_1(z - iv_2) \\ \vdots \\ \ell_m(z) = \prod_{i \in \mathbb{F}_p} \ell_{m-1}(z - iv_m). \end{cases}$$

The functions ℓ_j are examples of linearized polynomials, as previously defined. Our next step is to present the theorems from (PETIT, 2014). We suggest the reader to consult the original work for proofs and more details.

Theorem 4.1 *a) Each polynomial ℓ_i is split and its roots are all elements of the vector space generated by $\{v_1, \dots, v_i\}$. In particular, we have $\ell_n(z) = z^{p^m} - z$.*

b) We have $\ell_i(z) = \ell_{i-1}(z)^p - a_i \ell_{i-1}(z)$ where $a := (\ell_{i-1}(v_i))^{p-1}$.

c) If we identify $\mathbb{F}_{p^m}$ by the vector space $(\mathbb{F}_p)^m$, then each ℓ_i is a p-to-1 linear map of $\ell_{i-1}(z)$ and a p^i to 1 linear map of z.

From Theorem 4.1 and its properties, we can reach the following polynomial system:

$$\begin{cases} f(x_1) = 0 \\ x_j^p = a_j x_j = x_{j+1} \quad j = 1, \dots, m-1 \\ x_n^p - a_n x_n = 0 \end{cases} \tag{4.11}$$

where the $a_i \in \mathbb{F}_{p^n}$ are defined as in Theorem 4.1. Any solution of this system provides us with a root of f by the first equation, and the n last equations together imply this root belongs to $\mathbb{F}_{p^n}$. From this system of equations, (PETIT, 2014) derives Theorem 4.2.

Algorithm 10: Linearized polynomials for finding roots over $\mathbb{F}_{2^m}$.

Data: F as a univariate polynomial over $\mathbb{F}_{2^m}$ with degree t and m the extension field degree.

Result: R as a set of roots of p.

1 $\ell_i^k \leftarrow \emptyset$; $\ell_{is} \leftarrow \emptyset$; $A_k^j \leftarrow \emptyset$; $R \leftarrow \emptyset$; $dummy \leftarrow \emptyset$;
2 **if** $f_0 == 0$ **then**
3 $\quad R.append(0)$;
4 **end**
5 **for** $i \leftarrow 0$ **to** $\lceil (t-4)/5 \rceil$ **do**
6 $\quad \ell_i(x) \leftarrow 0$;
7 $\quad$ **for** $j \leftarrow 0$ **to** 3 **do**
8 $\quad\quad \ell_i(x) \leftarrow \ell_i(x) + f_{5i+2^j} x^{2^j}$;
9 $\quad$ **end**
10 $\quad \ell_{is}[i] \leftarrow \ell_i(x)$;
11 **end**
12 **for** $k \leftarrow 0$ **to** $m-1$ **do**
13 $\quad$ **for** $i \leftarrow 0$ **to** $\lceil (t-4)/5 \rceil$ **do**
14 $\quad\quad \ell_i^k \leftarrow \ell_{is}(\alpha^k)$;
15 $\quad$ **end**
16 **end**
17 $A_i^0 \leftarrow \emptyset$;
18 **for** $i \leftarrow 0$ **to** $\lceil (t-4)/5 \rceil$ **do**
19 $\quad A_i^0 \leftarrow f_{5i}$;
20 **end**
21 $X \leftarrow$ generate(m);
22 **for** $j \leftarrow 1$ **to** $2^m - 1$ **do**
23 $\quad$ **for** $i \leftarrow 0$ **to** $\lceil (t-4)/5 \rceil$ **do**
24 $\quad\quad A \leftarrow A_i^{j-1}$;
25 $\quad\quad A \leftarrow A + \ell_i^{\delta(X[j], X[j-1])}$;
26 $\quad\quad A_i^j \leftarrow A$;
27 $\quad$ **end**
28 **end**
29 **for** $j \leftarrow 1$ **to** $2^m - 1$ **do**
30 $\quad result \leftarrow 0$;
31 $\quad$ **for** $i \leftarrow 0$ **to** $\lceil (t-4)/5 \rceil$ **do**
32 $\quad\quad result = result + (X[j])^{5i} A_i^j$;
33 $\quad$ **end**
34 $\quad eval = result + f_3(X[j])^3$;
35 $\quad$ **if** $eval == 0$ **then**
36 $\quad\quad R.append(X[j])$;
37 $\quad$ **else**
38 $\quad\quad dummy.append(X[j])$;
39 $\quad$ **end**
40 **end**
41 **return** R;

Theorem 4.2 *Let $(x_1, x_2, \ldots, x_m)$ be a solution of the equations in Equation (4.11). Then $x_1 \in \mathbb{F}_{p^m}$ is a solution of f. Conversely, given a solution $x_1 \in \mathbb{F}_{p^m}$ of f, we can reconstruct a solution of all equations in Equation (4.11) by setting $x_2 = x_1^p - a_1 x_1$, etc.*

In (PETIT, 2014), the author presents an algorithm for solving the system in Equation (4.11) using resultants. The solutions of the system are the roots of a polynomial f. It is worth remarking that this algorithm is almost constant-time, and hence we just need to protect the branches presented on it.

4.5 RABIN ALGORITHM

Our last proposal for finding the roots of the error locator polynomial was the classical method proposed by Rabin (RABIN, 1980). This classical method is a probabilistic algorithm and has running time similar to the Cantor-Zassenhaus algorithm (CANTOR; ZASSENHAUS, 1981). This method is used to factor huge polynomials, with a higher degree and over larger fields. Although the main focus of this work relies on polynomials with no more than 200 roots and fields of size at most 2^{18}, we consider this algorithm because of its efficiency.

The Cantor-Zassenhaus differs from Rabin's method on the field extension of the polynomial to be factored. Since Cantor's method is designed only for odd extensions, we do not consider them in our works. However, Rabin made an adaptation to support even fields, introducing a trace computation to finding a nontrivial factorization of the input polynomial, similar to the BTA method. This addition increases the execution time of the algorithm, but it is still efficient, even when used in large field extensions.

The main difference between Cantor-Zassenhaus and Rabin's method to the other methods presented in this chapter is the insertion of a randomness choice of an element in the algorithm. The computation of the roots depends on the choice of a random element in $\mathbb{F}_{2^m}$. This random selection was used with a trace function and a *gcd* to find a nontrivial factorization of the original polynomial. This class of algorithm is also called probabilistic algorithms.

Rabin method is a probabilistic algorithm because of its random behavior, because of the chance of failure, the parameter ε is used to define the maximum number of iterations on the algorithm. Since we always choose a random element to continue, the probability of failure of the algorithm is based on the fact that a random selection could not result in a nontrivial factor of the input polynomial. However, Ben-Or proves that this probability is $1 - 1/2^{t-1}$, where t is the degree of the polynomial (BEN-OR, 1981).

To the best of our knowledge, no code-based cryptosystem makes use of a probabilistic algorithm as a method for root finding. The original proposal was designed as a recursive algorithm, here we present an iterative version, as show in (VON ZUR GATHEN; PANARIO, 2001). Algorithm 11 shows all steps to compute the roots.

Rabin's algorithm is an equal degree factorization method. Hence we are computing the roots of a polynomial that is composed only by linear factors. We simplified a step where

Algorithm 11: Probabilistic root finding algorithm – $Rabin(p)$.

```
   Data: p as an univariate polynomial over F_{2^m}, t as number of expected roots.
   Result: The set of roots of p.
 1 Factors ← {p}; // The set of polynomials to be computed
 2 k ← 1;
 3 it_max ← 2⌈log t^2/ε⌉;
 4 while k < it_max do
 5     h ←$ F_{2^m} // choose h with degree < t at random
 6     g ← gcd(h, f);
 7     if g = 1 then
 8         g ← tr(h)
 9     end
10     for u ∈ Factors  and  u.degree > d do
11         if gcd(g, u) ≠ 1  and  gcd(g, u) ≠ u then
12             Factors.remove(u);
13             Factors.insert(gcd(g, u));
14             Factors.insert(u/gcd(g, u));
15         end
16     end
17 end
18 return Factors
```

the original method computes before the trace $g = \sum_{i=0}^{d-1} h^{q^i}$, where d is equal to the degree of the factors of f.

Rabin's algorithm is a great example of a naturally safe root-finding against timing side-channel attack. Since it uses a random selection on their execution, we consider that this randomness does not permit an attacker to infer any information about the roots. Moreover, since we are using a random selection, an attacker could not infer any information from the time variance of the algorithm. A more detailed analysis was presented in Chapter 5.

5 COMPARISON

In this chapter, we present an analysis of the five presented methods in the Chapter 4. The first two analyses focus on the complexity and performance of the algorithms. However, we are not interested only in efficient techniques. Our primary goal was to achieve a method with no information leakage against a timing side-channel attack. Hence, we demonstrate a time-variance analysis for each proposed root-finding method. After that, we present a security analysis of the algorithm. We remark that $n = 2^m$ is the size fields, and t is the degree of the polynomial.

5.1 COMPLEXITY ANALYSIS

In order to compare the complexity of the algorithm, we use the Big $\mathcal{O}$ notation. This asymptotic notation permits us to classify the algorithms according to their behavior when the inputs grow towards infinity.

Table 1 – Complexity comparison

Method	
Exhaustive search	$\mathcal{O}(t2^m)$
Permuted exhaustive	$\mathcal{O}(t2^m)$
Linearized Polynomials	$\mathcal{O}(2^m)$
Constant Linearized Polynomials	$\mathcal{O}(2^m)$
Berlekamp trace algorithm	$\mathcal{O}(t^2m)$
Iterative Berlekamp trace algorithm	$\mathcal{O}(m^2t^2)$
Successive resultant algorithm	$\mathcal{O}(t^2m^3)$
Constant Successive resultant algorithm	$\mathcal{O}(t^2m^3)$
Rabin root finding	$\mathcal{O}(t^2m)$

As we can note, the most asymptotic efficient method was the Rabin root-finding method and the Berlekamp trace algorithm. However, this asymptotic could not reflect the execution timing of the algorithm for parameters used in code-based schemes. This happens because the big $\mathcal{O}$ notation does not consider constants and the asymptotic complexity was for all m and t greater then m_0 and t_0 respectively, and the parameters used on code-based cryptosystems could be smaller than the m_0 and t_0.

5.2 PERFORMANCE ANALYSIS

To give a more careful analysis, we present an execution time analysis of each algorithm. This comparison gives an idea of the real execution cost. All root finding-methods were performed over a random polynomial of degree t over $\mathbb{F}_{2^m}$, for $t = 50$ until $t = 200$ and $n = 2^{10}$ to $n = 2^{16}$. For each polynomial, we take the average time of 10 executions to minimize the environment noise. Figure 2 show the obtained results.

The performance analysis was implemented using SageMath (The Sage Developers, 2020), since we want to measure the execution cost with different parameters, the Sage library permits us to change the field with an easy way to perform this analysis. We run all proposed methods in a locally Intel® Core(TM) i7-4500U CPU @ 1.80GHz. The source code of this attack are available on https://github.com/doodomartins/root-finding/soft-factorization.

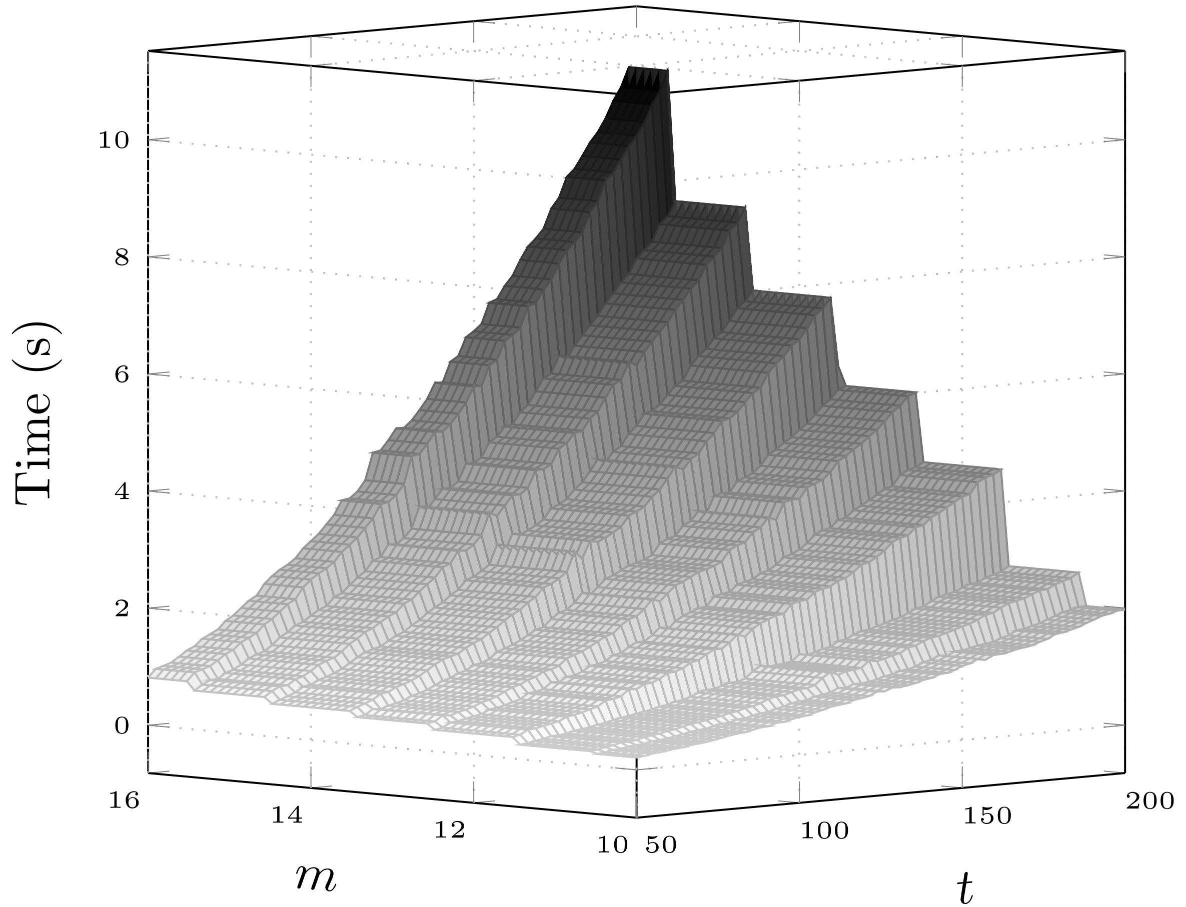
The Figure 2d to Figure 2c presents the execution time for each root finding method that has exponential behaviour. For each of that algorithm, we can observe the same behavior on execution time. Since the exponential term in the complexity was the extension m, we can note that the algorithms have a different growth for each extension. The higher m results in a growth on the runtime.

Moreover, we can observe the massive difference between the extension 2^{15} to 2^{16} on the exhaustive search and in the linearized algorithm. On the other hand, the degree of the polynomial affects the execution time of the algorithm linearly. Therefore, the execution time grows more slowly when we are increasing the degree of the polynomial.

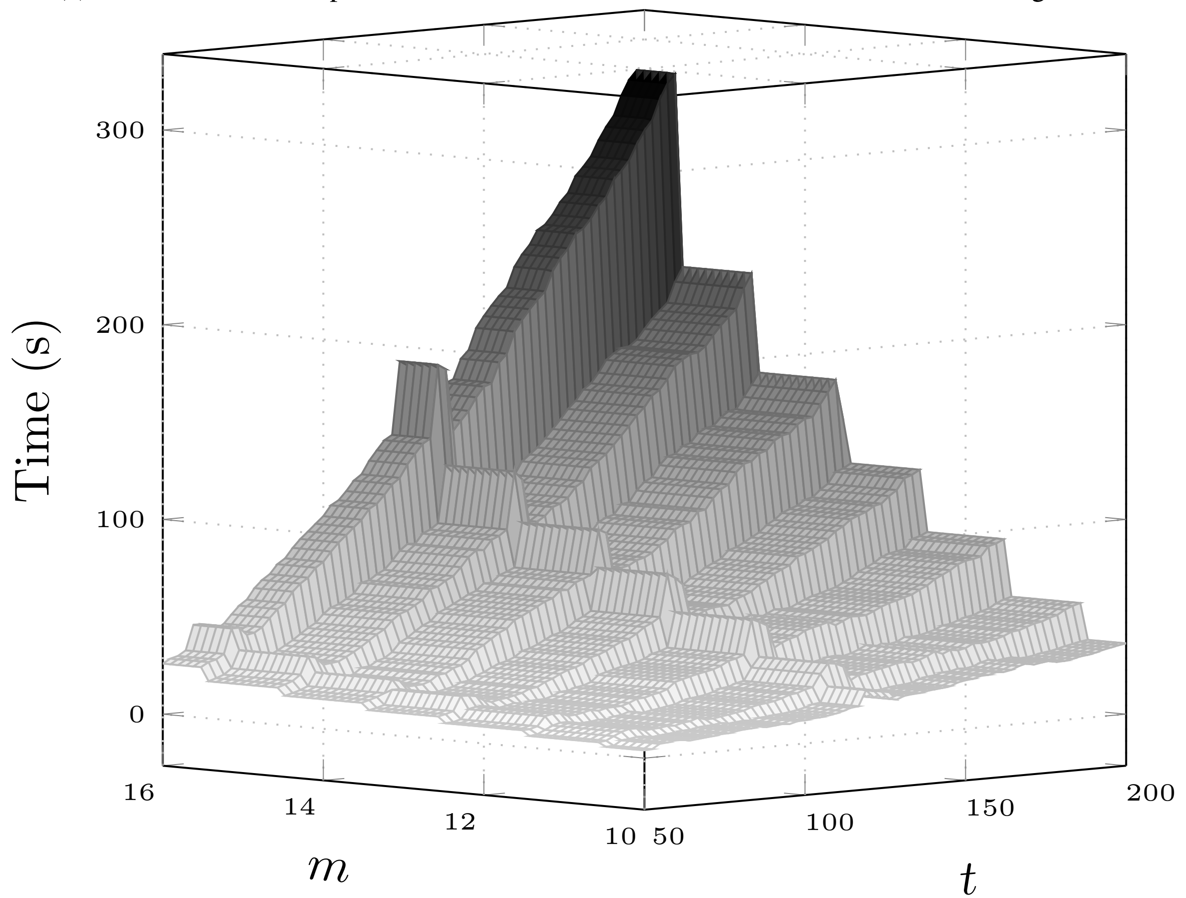
For Successive Resultant Algorithm and the Berlekamp Trace algorithm we can note a suitable growth in relation to the field extension m, Figure 2a and Figure 2b. This relies that both of then are linearly related with the field extension. As we expect in all method, SRA and BTA execution time also increase when we increase the degree of the input polynomial.

By last, the Rabin Factorization Algorithm has the most interesting behaviour when we compare different executions. Since we are analyzing an probabilistic algorithm, each execution of it depends directly to the random decision taken inside the algorithm. It is well know that this algorithm respect an linear growth curve. However, when we compare different execution with polynomials with close degree we can note that some higher polynomials has shorter run time then lower polynomials.

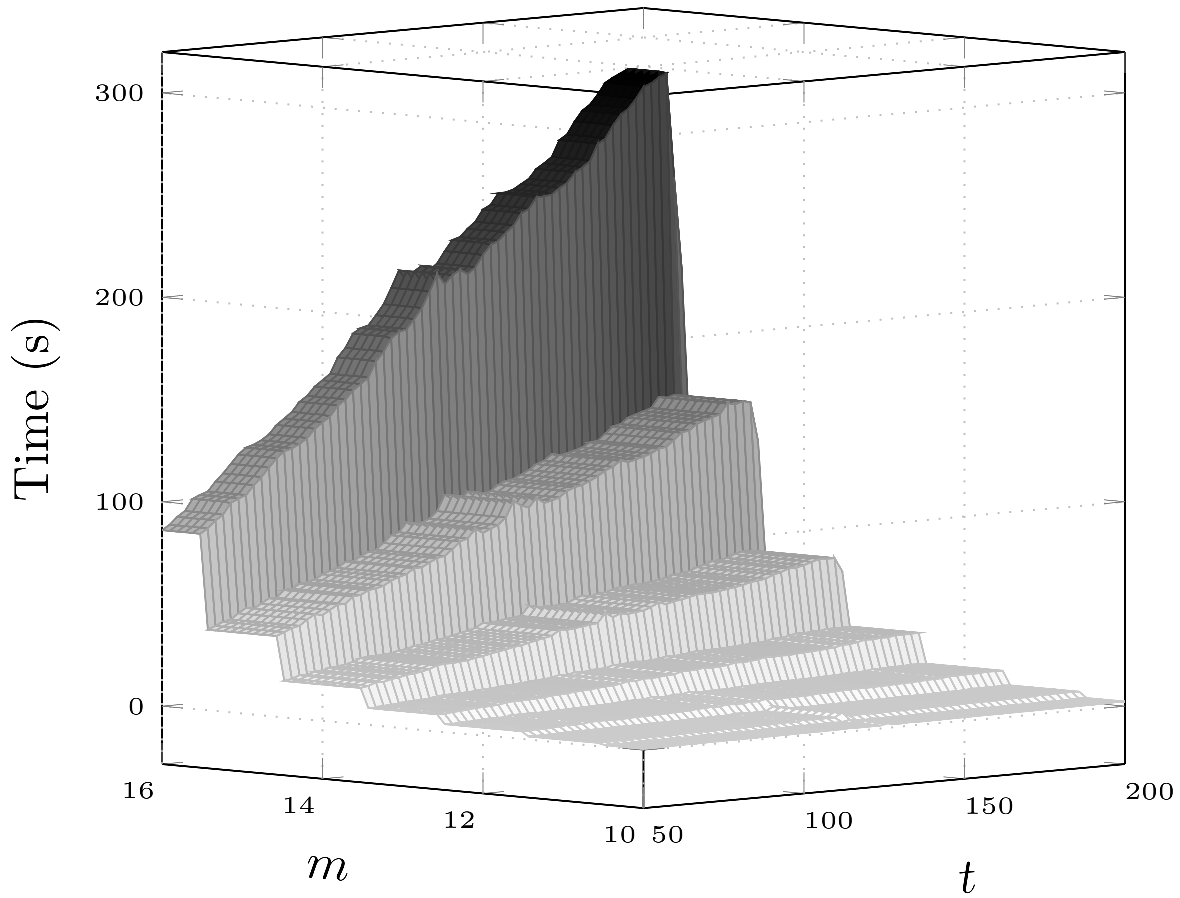
In our experiment for the Rabin method, we select $\varepsilon = 0.01$. And for all of our execution, the algorithm any returns a failure. Set the probability of failure closes to zero, permits to us find an scenario that our probabilistic algorithm has a behaviour closes to our deterministic algorithms. Also in our experiments, we note that the Rabin method starts to has a consider number of failures if we considerably reduce the ε value and increase the polynomial degree.



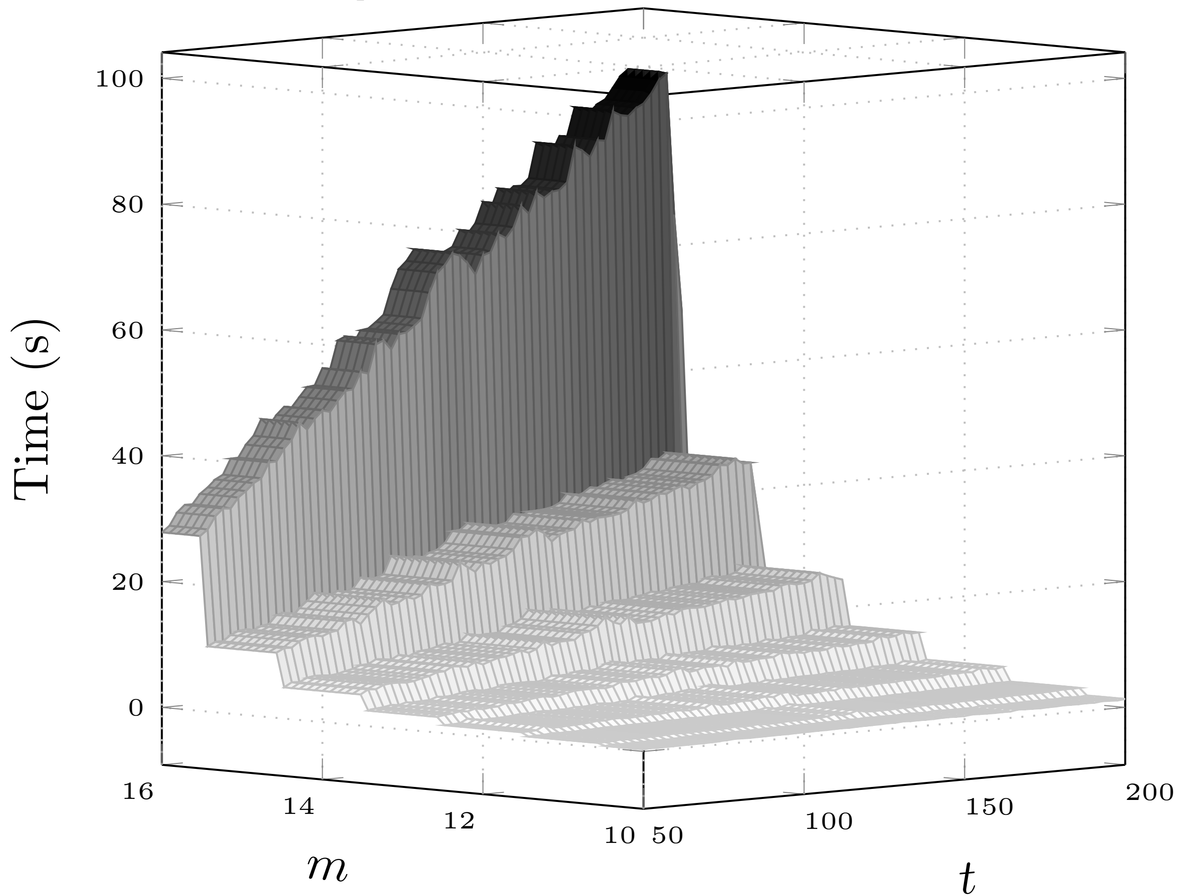
(a) Execution time comparison between different *m* and *t* for Successive resultant algorithm.



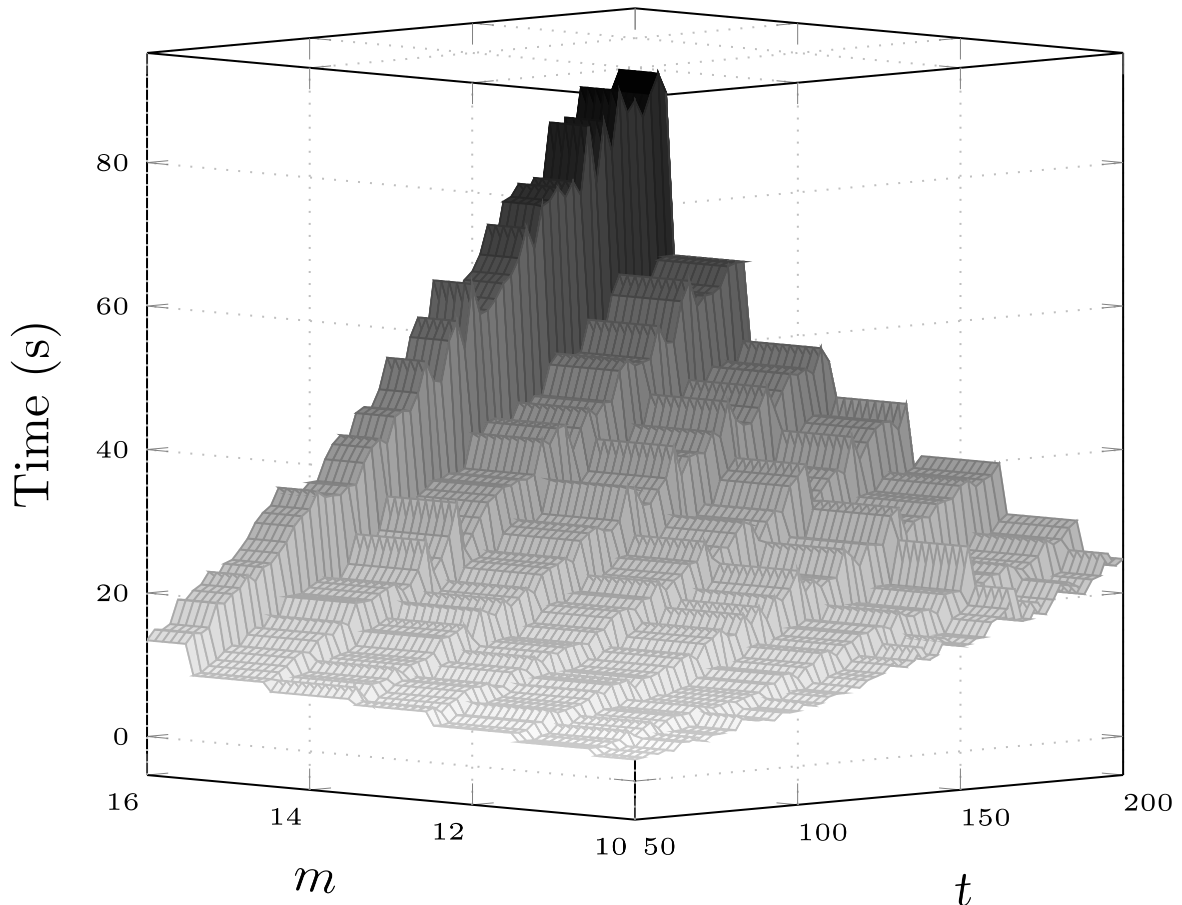
(b) Execution time comparison between different *m* and *t* for Berlekamp trace algorithm.



(c) Execution time comparison between different m and t for exhaustive search method.



(d) Execution time comparison between different m and t for linearized algorithm.



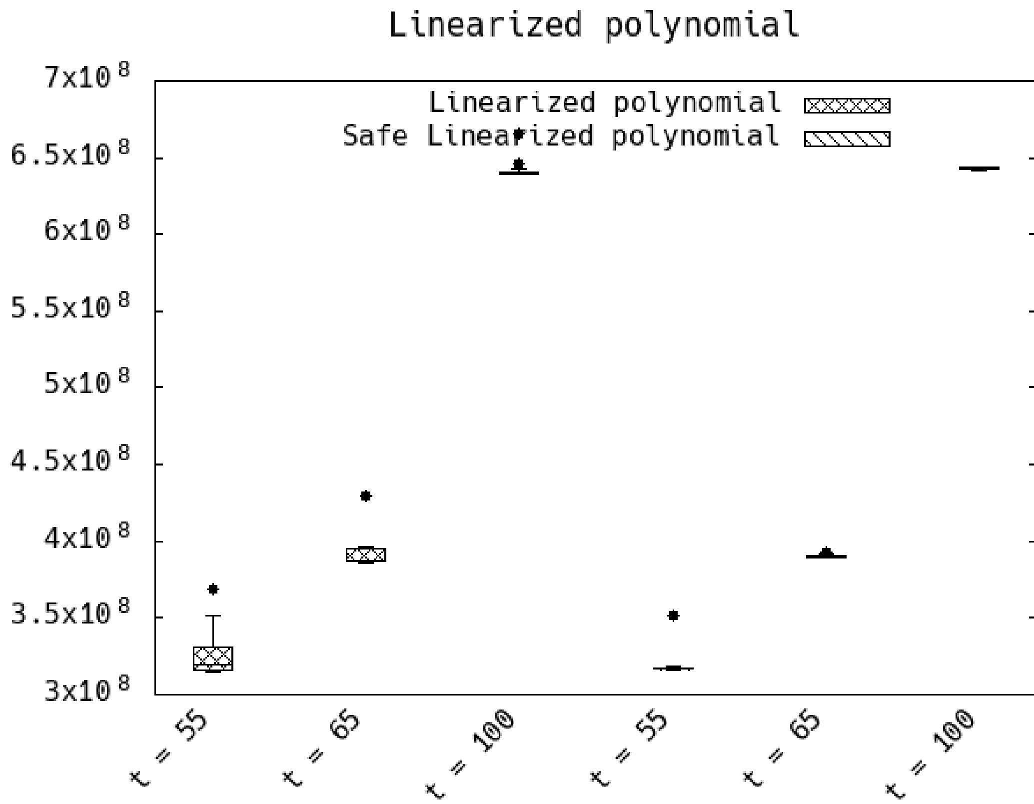
(e) Execution time comparison between different m and t for Rabin algorithm.

Figure 2 – Measurements time for the five methods presented. For the degree of the polynomial as $t = 50$ until $t = 200$ and the size of the field as $n = 2^{10}$ to $n = 2^{16}$.

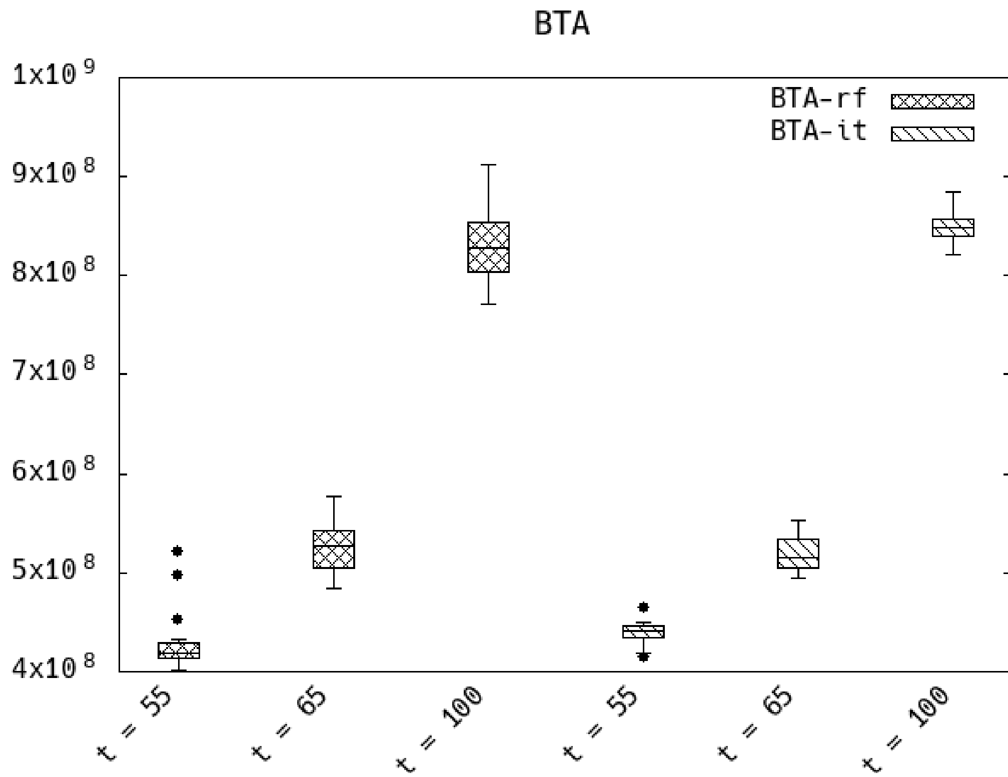
5.3 TIME VARIANCE ANALYSIS

The main focus of our work relies on propose alternatives to compute roots of error locator polynomial without leaking sensitive information against a timing side-channel attack. As previously presented, a naive implementation of the root-finding process allows an attacker to recover the ciphertext and compute a session key of a code-based cryptosystem. In order to avoid this attack, we present five algorithms to compute the roots of a polynomial over binary finite fields. For three of these algorithms, we propose countermeasures to reduce the time variance in its runtime. In this section, we present a time variance analysis of these countermeasures.

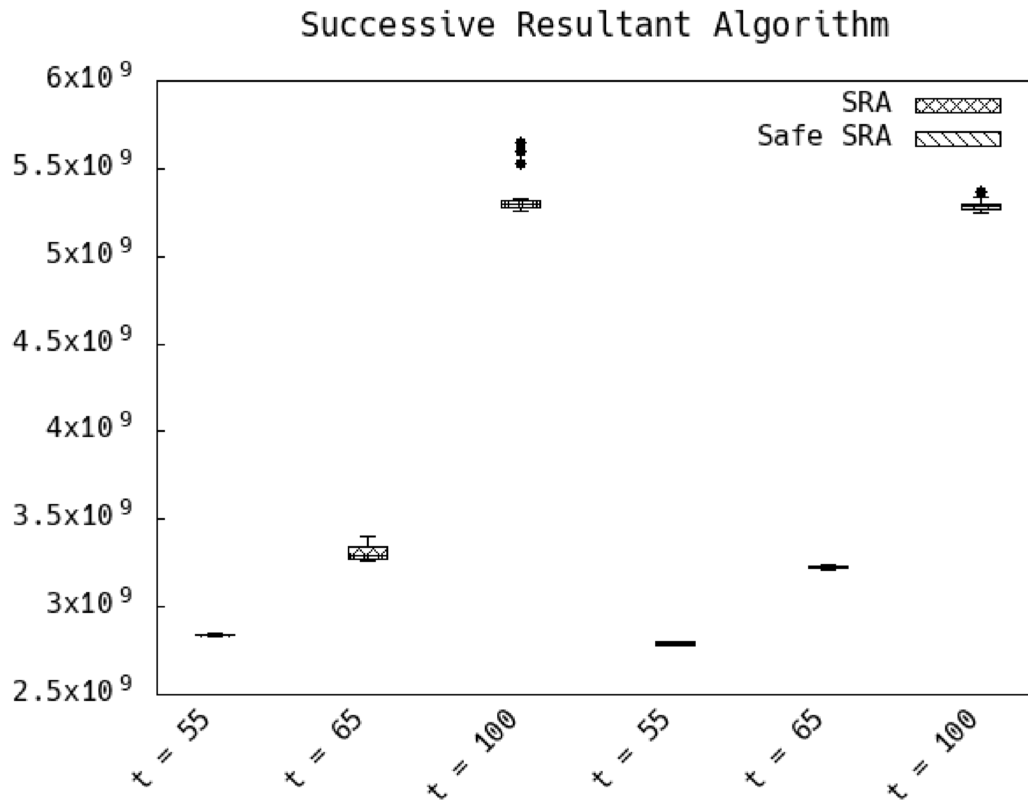
In order to measure the time variance, we perform the root-finding procedures and measure the time variance between different executions. This variance is measured in CPU cycles. Figure 3 presents the time variance for Berlekamp Trace Algorithm, Linearized Algorithm and Successive Resultant Algorithm for a polynomial with degree $t = \{55, 60, 100\}$ and $\mathbb{F}_{2^{16}}$.



(a) Comparison between linearized polynomials with and without countermeasures.



(b) Comparison between BTA-rf and BTA-it executions.



(c) Comparison between SRA and Safe SRA executions.

Figure 3 – Measurements cycles for methods presented in Chapter 4. Our evaluation of SRA was made using a Python implementation and cycles measurement with C. In our tests, the drawback of calling a Python module from C has behavior bordering to constant.

For all measures in Figure 3, we could note that we reduce the time variance to compute the roots of different polynomials with same degree. In specific, on Linearized polynomial comparison, we can note that all quartiles are closer to the average value. This behaviour also is noticed for on Successive Resultant Algorithm and even more on BTA.

An important difference in our implementation and the original was that we increase the execution time for all executions. This is justified by all extra operations that we add in order to reduce the time variance. However, we also can note that this growth is not too significant. Since we still are close to the original values. A more detailed comparison, for only polynomials with degree 100 is presented in Figure 4.

For the execution with degree equals to 100, we can observe the same behaviour as with other degrees. In all methods, when we apply our countermeasure, we increase the average execution time. However, we can note that we considerably reduced the time variance of the algorithm. This variance can be observed on the quartiles in Figure 4. Since the quartiles inside the box represents 50% of all measured time, and the other quartiles represent the remained 50%, ignoring the outliers, we can uphold that our countermeasures reduce the time variance of the root-finding methods.

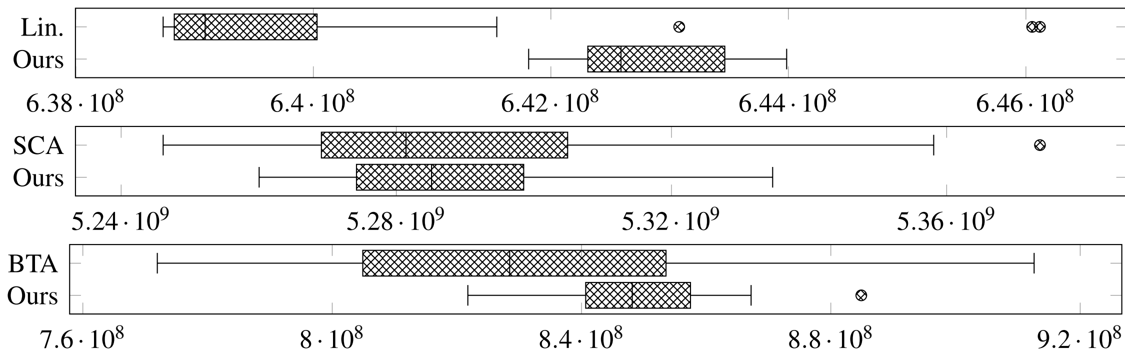


Figure 4 – Comparison of original implementation and our proposal for Linearized Algorithm - Lin., Successive resultant algorithm - SCA., and Berlekamp trace algorithm - BTA with $t = 100$.

A notable time variation reduction can be viewed on BTA comparison. We obtain this huge variation reduction because we completely redesign the algorithm structure. Now the algorithm works always dividing the currently polynomial into two polynomials with an approximated degree. This strategy results in a considerably time variation result. And, as expected, we increase the average execution time.

Another important improvement in our implementation was the reduction of outliers. A more specialized attacker, that is able to modify ciphertexts that result in an error locator polynomial with an execution time that is an outlier, could not more detect this information with our implementation.

5.4 SECURITY OVERVIEW

In order to present a security analysis over our countermeasures showed in Chapter 4, in this section we provide a general security overview of our proposals. Our main hypothesis on the BIGQUAKE attack was the fact that the time variance on the root-finding process leak information about which kind of polynomial was factorized. This approach leads us to recover a cipher message in a few minutes on a normal laptop. Nonetheless, this attack is hard to be implemented in real cases. The main problem of this attack in a key encapsulation mechanism protocol relies on that one of the parts of the protocol could abort after a few attempts and restart the protocol.

Nonetheless, an attacker with a more precise method to measure the decryption time could be able to guess part of the original plain message and with large computational power could be enabled to make an educated guess to find the correct session key. In order to increase the difficulty to correctly perform a timing side-channel attack over McEliece schemes, we presented five countermeasures to avoid this information leakage.

The first countermeasure is based on the direct method, the Exhaustive Search. This method was quite simple to be implemented. With the permutation applied to the elements

before the execution of all evaluations, we shuffle all elements taking out the chance of an attacker guess a root based on when an extra operation is executed.

The main drawback of this method was your exponential behaviour. Since an attacker could modify an encrypted message to add more errors, the exhaustive search will also increase its execution time. Despised this time variance it will not be huge, an attacker with a delicate measure tool will notice this variance and still will acquire information.

Our second countermeasure is based on reducing the time variance of the algorithms. This strategy is applied to Berlekamp Trace Algorithm, Linearized Algorithm, and Successive Resultant Algorithm. The main idea of this countermeasure was to present a protected implementation where the time difference between executions will be kept to the minimum.

The first countermeasure, on Berlekamp's method, consists of change the recursive classical implementation to an iterative one. Next, we apply the same techniques in all three methods in order to achieve a reduction in the time variance of each algorithm. This simple strategy was used to protect the implementation against a timing side-channel attacker. We considerably reduce the time variance and this reduction is presented on experiments results in Section 5.3.

With a smaller time variance on the root-finding algorithm implementation, we can reduce the association between the time execution and the input polynomial. Consequently, reducing the relation with the ciphertext. With our proposal we can build a more secure implementation of the McEliece decryption.

Our last countermeasure was the usage of a probabilistic algorithm. The Rabin's method execution time is related to a random choice of a polynomial. Each iteration of the algorithm has a different random selection. This randomized behavior results in different execution times for each different execution. Even more, we cannot determine if the time variance of the algorithm between two executions relies upon the characteristics of the input polynomial or for the random selections.

The idea of use a probabilistic algorithm to finding roots in code-based cryptosystems still was not exhaustively studied in the literature. However, we consider its use secure, because, in our tests, we could control the fail rate by selecting a total number of iteration that permit to us reduce the failure rate to zero. Thus, an attacker measuring the time variance of this algorithm cannot access any sensitive information of the plain text message of the code-based cryptosystem.

6 FINAL CONSIDERATIONS

In this thesis, we propose countermeasures on root-finding algorithms in order to achieve a decoding process without leaking any sensitive information against a timing side-channel attack. We propose five methods, with different characteristics that can be applied to the root extraction task. Our proposals are based on reducing the time variance by applying implementation techniques which aim to construct an algorithm without branches and with constant behavior.

Before presenting our countermeasures, we execute a timing attack against a Round 1 NIST proposal, the BIGQUAKE submission. This attack uses the fact that a naive implementation on the decoding process, more specifically, on the root-finding method, leaks information about the polynomial that has been factorized and consequently about the error added to the message.

The attack is presented to illustrate how insecure a naive implementation is. In our experiments, we detect that the root-finding method was responsible for the major time variance on a root-finding algorithm. Thus, we present five alternatives to construct secure decoding. Our countermeasures are based on Exhaustive search, the Berlekamp Trace Algorithm, the Linearized Algorithm, the Successive Resultant Algorithm, and the Rabin Algorithm.

These countermeasures were implemented and analyzed in order to measure their time variance and their behavior. In our analysis, we observe that we reduce the time variance for all methods where we propose to reduce this variance. Also, in the probabilistic method, we note that the behavior of the original method does not leak any information about the polynomial that has been factorized. This dissimilarity relies on the fact that the algorithm takes a random selection in each iteration.

We recommend the usage of three root-finding methods presented in this works. First, the use of the new iterative Berlekamp trace algorithm; in our experiments, the BTA method presents a small-time variance. However, it uses the Euclidean algorithm to compute the *gcd* in their execution, and it is well known in the literature that the classical implementation of *gcd* has no constant time. Second, the usage of the SRA approach, with our countermeasures, also presents a small-time variance; however, its implementation was more complicated because it relies on multivariate polynomial systems.

Lastly, we suggest the usage of the Rabin Algorithm, since it has a random behavior, with no way to infer information from the execution time. This was easily observed on the time execution analysis presented in Figure 2e. However, the *gcd* algorithm, as in BTA could be used to employ a timing attack. Notwithstanding, to the best of our knowledge, any existent timing attack on the literature can infer sufficient information of the *gcd* to acquire information of the polynomial that was being factorized.

Two of our proposed countermeasures, the Berlekamp algorithm, and the Rabin Algorithm, we don't solve the problem of the non-constant implementation of the *gcd*. It is well known that its implementation could be insecure, for thus we strongly suggest the study of new

methods to achieve a secure implementation of the *gcd*. One of the candidates for a secure implementation is the recent work from Bernstein and Yang (BERNSTEIN; YANG, 2019). This new work presents a new variant to compute the Euclidean algorithm for polynomials and integers that aims to have constant execution time.

Using this new approach, we can improve two of our proposed methods. First, on BTA, we can deploy a method that will have in all your steps a constant behaviour. With all branches having the same coast on final execution time. The second improvement that the constant *gcd* gives to us is a more random behaviour com Rabin's method. With this new feature, the random selection will be mandatory on the total execution time, removing totally the relation between the execution time and the input polynomial.

6.1 FUTURE WORK

For future work, we suggest the following:

- Employ a constant-time *gcd* to our implementation: Bernstein and Yang recently propose a constant time approach to perform the *gcd* algorithm (BERNSTEIN; YANG, 2019). This new approach claims to solve the problem of a non-constant *gcd* that our root-finding methods present.
- Use a dedicated environment to perform a more detailed analysis: Our implementation was made on a common laptop with a general propose operation system. This implies that other applications were running during the execution of our experiments. To mitigate this, we run ten times each execution and take the average time. However, a dedicate environment can present more precise results to measure the time variance.
- We bring the attention of the reader that we did not use any optimization in our implementations, i.e., we did not use vectorization or bit slicing techniques or any specific instructions such as Intel® IPP Cryptography for finite field arithmetic in our code. Therefore, these techniques and instructions can improve the finite field operations and speed up our algorithms.
- We remark that for achieving a safer implementation, one needs to improve the security analysis, by removing conditional memory access and protecting memory access of instructions. Moreover, one can analyze the security of the implementations by considering different attack scenarios and performing an in-depth analysis of hardware side-channel attacks.

BIBLIOGRAPHY

BARDET, M. et al. Big quake: Binary goppa quasi–cyclic key encapsulation. **NIST submissions**, 2017.

BARRETO, P. S. L. M.; LINDNER, R.; MISOCZKI, R. **Decoding square-free Goppa codes over Fp**. 2010. Cryptology ePrint Archive, Report 2010/372. Https://eprint.iacr.org/2010/372.

BEN-OR, M. Probabilistic algorithms in finite fields. In: IEEE. **22nd Annual Symposium on Foundations of Computer Science (1981)**. [S.l.], 1981. p. 394–398.

BERLEKAMP, E.; MCELIECE, R.; TILBORG, H. V. On the inherent intractability of certain coding problems. **IEEE Transactions on Information Theory**, v. 24, n. 3, p. 384–386, 1978.

BERLEKAMP, E. R. Factoring polynomials over large finite fields. **Mathematics of Computation**, v. 24, n. 111, p. 713–735, 1970.

BERNSTEIN, D. J. **Cache-timing attacks on AES**. 2005. (accessed 2020-01-11) Available in: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

BERNSTEIN, D. J. et al. Classic McEliece: conservative code-based cryptography. **NIST submissions**, 2017.

BERNSTEIN, D. J.; CHOU, T.; SCHWABE, P. McBits: fast constant-time code-based cryptography. In: SPRINGER. **International Workshop on Cryptographic Hardware and Embedded Systems**. [S.l.], 2013. p. 250–272.

BERNSTEIN, D. J.; LANGE, T.; PETERS, C. Attacking and defending the McEliece cryptosystem. In: SPRINGER. **International Workshop on Post-Quantum Cryptography**. [S.l.], 2008. p. 31–46.

BERNSTEIN, D. J.; PERSICHETTI, E. **Towards KEM Unification**. 2018. Cryptology ePrint Archive, Report 2018/526. Https://eprint.iacr.org/2018/526.

BERNSTEIN, D. J.; YANG, B.-Y. Fast constant-time gcd computation and modular inversion. **IACR Transactions on Cryptographic Hardware and Embedded Systems**, v. 2019, n. 3, p. 340–398, May 2019.

BISWAS, B.; HERBERT, V. Efficient root finding of polynomials over fields of characteristic 2. Https://hal.archives-ouvertes.fr/hal-00626997. 2009.

BLACK, P. E. Fisher-Yates shuffle – dictionary of algorithms and data structures. (accessed 2019-12-20) Available in: https://www.nist.gov/dads/HTML/fisherYatesShuffle.html. 1998.

BRUINDERINK, L. G. et al. Flush, Gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In: **Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings**. [S.l.: s.n.], 2016. p. 323–345.

BUCERZAN, D. et al. Improved timing attacks against the secret permutation in the McEliece PKC. **International Journal of Computers Communications & Control**, v. 12, n. 1, p. 7–25, 2017.

C++ Standards Committee. **ISO International Standard ISO/IEC 14882: 2014, Programming Language C++**. [S.l.], 2014.

CANTEAUT, A.; SENDRIER, N. Cryptanalysis of the original McEliece cryptosystem. In: SPRINGER. **International Conference on the Theory and Application of Cryptology and Information Security**. [S.l.], 1998. p. 187–199.

CANTOR, D. G.; ZASSENHAUS, H. A new algorithm for factoring polynomials over finite fields. **Mathematics of Computation**, p. 587–592, 1981.

CHIEN, R. Cyclic decoding procedures for bose-chaudhuri-hocquenghem codes. **IEEE Transactions on information theory**, v. 10, n. 4, p. 357–363, 1964.

CHOU, T. Mcbits revisited. In: **Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings**. [S.l.: s.n.], 2017. p. 213–231.

DAEMEN, J.; RIJMEN, V. The design of rijndael: AES-the advanced encryption standard. **Springer Science & Business Media**, 2013.

DAVENPORT, J. H.; PETIT, C.; PRING, B. A Generalised Successive Resultants Algorithm. In: DUQUESNE, S.; PETKOVA-NIKOVA, S. (Ed.). **Arithmetic of Finite Fields**. [S.l.: s.n.], 2016. p. 105–124.

DIFFIE, W.; HELLMAN, M. New directions in cryptography. **IEEE Transactions on Information Theory**, v. 22, n. 6, p. 644–654, 1976.

FAUGERE, J.-C. et al. A distinguisher for high-rate McEliece cryptosystems. **IEEE Transactions on Information Theory**, v. 59, n. 10, p. 6830–6844, 2013.

FEDORENKO, S. V.; TRIFONOV, P. V. Finding roots of polynomials over finite fields. **IEEE Transactions on Communications**, v. 50, n. 11, p. 1709–1711, 2002.

KOCHER, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: **Annual International Cryptology Conference**. [S.l.: s.n.], 1996. p. 104–113.

MCELIECE, R. J. A Public-Key Cryptosystem Based On Algebraic Coding Theory. **Deep Space Network Progress Report**, v. 44, p. 114–116, jan. 1978.

NIEDERREITER, H. Knapsack-type cryptosystems and algebraic coding theory. **Problems of Control and Information Theory. Problemy Upravlenija i Teorii Informacii**, v. 15, n. 2, p. 159–166, 1986.

PATTERSON, N. The algebraic decoding of Goppa codes. **IEEE Transactions on Information Theory**, v. 21, n. 2, p. 203–207, 1975.

PETIT, C. Finding roots in GF(p^n) with the successive resultant algorithm. **IACR Cryptology ePrint Archive**, v. 2014, p. 506, 2014.

RABIN, M. O. Probabilistic algorithms in finite fields. **SIAM Journal on computing**, v. 9, n. 2, p. 273–280, 1980.

RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. **Communications of the ACM**, v. 21, n. 2, p. 120–126, 1978.

SAVAGE, C. A survey of combinatorial Gray codes. **SIAM Review**, v. 39, n. 4, p. 605–629, 1997.

SHOR, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. **SIAM J. Comput.**, v. 26, n. 5, p. 1484–1509, out. 1997. ISSN 0097-5397.

SHOUFAN, A. et al. A timing attack against patterson algorithm in the McEliece PKC. In: **Information, Security and Cryptology - ICISC 2009, 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers**. [S.l.: s.n.], 2009. p. 161–175.

SKACHEK, V.; ROTH, R. M. Probabilistic algorithm for finding roots of linearized polynomials. **Designs, Codes and Cryptography**, v. 46, n. 1, p. 17–23, Jan 2008.

STRENZKE, F. Fast and secure root finding for code-based cryptosystems. In: **International Conference on Cryptology and Network Security**. [S.l.: s.n.], 2012. p. 232–246.

STRENZKE, F. **Efficiency and implementation security of code-based cryptosystems**. [S.l.]: Technische Universität, 2013. Phd Thesis.

The Sage Developers. **SageMath, the Sage Mathematics Software System (Version 9.0.0)**. [S.l.], 2020. `https://www.sagemath.org`.

TRUONG, T.-K.; JENG, J.-H.; REED, I. S. Fast algorithm for computing the roots of error locator polynomials up to degree 11 in Reed-Solomon decoders. **IEEE Transactions on Communications**, v. 49, n. 5, p. 779–783, 2001.

VON ZUR GATHEN, J.; PANARIO, D. Factoring polynomials over finite fields: A survey. **Journal of Symbolic Computation**, v. 31, n. 1-2, p. 3–17, 2001.

WANG, W.; SZEFER, J.; NIEDERHAGEN, R. FPGA-based Niederreiter cryptosystem using binary goppa codes. In: **Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings**. [S.l.: s.n.], 2018. p. 77–98.

A IMPLEMENTATION REMARKS ON BIGQUAKE ATTACK

Listing A.1 – Code snippet of BIGQUAKE error attribution

```
//Construct an error e from m
int error = (int *) malloc(NB_ERRORS*sizeof(int));
m2errir(m, error);

//Encrypt e (Niederreiter)
unsigned char* syndrome =
  malloc(SYNDROME_BYTES*sizeof(unsigned char));
decrypt_nied(syndrome, error, (unsigned char*) sk);
```

Listing A.2 – Code snippet of BIGQUAKE error attribution with the fix.

```
//Construct an error e from m
int error = {0};
m2errir(m, error);

//Encrypt e (Niederreiter)
unsigned char* syndrome =
  malloc(SYNDROME_BYTES*sizeof(unsigned char));
decrypt_nied(syndrome, error, (unsigned char*) sk);
```

Listing A.3 – Multiplication of two elements in $\mathbb{F}_{2^{12}}$ and inversion of an element in $\mathbb{F}_{2^{12}}$

```
#include <stdint.h>
typedef uint16_t gf;

//Multiplication between in0 and in1
gf gf_q_m_mult(gf in0, gf in1) {
    uint64_t i, tmp, t0 = in0, t1 = in1;
    //Multiplication
    tmp = t0 * (t1 & 1);
    for (i = 1; i < 12; i++)
        tmp ^= (t0 * (t1 & (1 << i)));
    //reduction
    tmp = tmp & 0x7FFFFF;
    //first step of reduction
    gf reduction = (tmp >> 12);
    tmp = tmp & 0xFFF;
    tmp = tmp ^ (reduction << 6);
```

```
    tmp = tmp ^ (reduction << 4);
    tmp = tmp ^ reduction << 1;
    tmp = tmp ^ reduction;
    //second step of reduction
    reduction = (tmp >> 12);
    tmp = tmp ^ (reduction << 6);
    tmp = tmp ^ (reduction << 4);
    tmp = tmp ^ reduction << 1;
    tmp = tmp ^ reduction;
    tmp = tmp & 0xFFF;
    return tmp;
}
```

Listing A.4 – Inversion of an element in $\mathbb{F}_{2^{12}}$

```
#include <stdint.h>
typedef uint16_t gf;

// Multiplicative inverse of in
gf gf_inv(gf in) {
    gf tmp_11 = 0;
    gf tmp_1111 = 0;
    gf out = in;
    out = gf_sq(out); //a^2
    tmp_11 = gf_mult(out, in); //a^2*a = a^3
    out = gf_sq(tmp_11); //(a^3)^2 = a^6
    out = gf_sq(out); // (a^6)^2 = a^12
    tmp_1111 = gf_mult(out, tmp_11); //a^12*a^3 = a^15
    out = gf_sq(tmp_1111); //(a^15)^2 = a^30
    out = gf_sq(out); //(a^30)^2 = a^60
    out = gf_sq(out); //(a^60)^2 = a^120
    out = gf_sq(out); //(a^120)^2 = a^240
    out = gf_mult(out, tmp_1111); //a^240*a^15 = a^255
    out = gf_sq(out); // (a^255)^2 = 510
    out = gf_sq(out); //(a^510)^2 =  1020
    out = gf_mult(out, tmp_11); //a^1020*a^3 = 1023
    out = gf_sq(out); //(a^1023)^2 = 2046
    out = gf_mult(out, in); //a^2046*a = 2047
    out = gf_sq(out); //(a^2047)^2 = 4094
    return out;
}
```